TRƯỜNG ĐẠI HỌC ĐÀ LẠT
KHOA TOÁN - TIN HỌC

TRƯƠNG CHÍ TÍN

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT 1

(Giáo Trình)

-- Lưu hành nội bộ --

Đà Lạt 2008

LỜI MỞ ĐẦU

Giáo trình này nhằm cung cấp cho sinh viên các kiến thức căn bản về các cấu trúc dữ liệu cơ sở có cấu trúc tuyến tính tĩnh, động (danh sách liên kết), cấu trúc cây và các giải thuật cơ bản liên quan đến chúng như sắp xếp, tìm kiếm ở bộ nhớ trong, cũng như so sánh độ phức tạp của các giải thuật này. Để có thể nắm bắt các kiến thức trình bày học phần này, sinh viên cần nắm được các kiến thức về tin học đại cương, nhập môn lập trình. Ngôn ngữ lập trình được chọn để minh họa các kiến thức trên là C++. Các kiến thức này sẽ tạo điều kiện cho học viên tiếp tục dễ dàng nắm bắt các kiến thức các học phần tin học về sau như: cấu trúc dữ liệu và giải thuật nâng cao, phân tích và thiết kế giải thuật, đồ hoạ, hệ điều hành, trí tuệ nhân tạo, ...

Nội dung giáo trình gồm 4 chương:

- Chương 1: Giới thiệu các khái niệm ban đầu về mối liên hệ mật thiết giữa cấu trúc dữ liệu và giải thuật, kiểu dữ liệu, thiết kế và phân tích giải thuật, độ phức tạp giải thuật, ...
- Chương 2: Giới thiệu các phương pháp cơ bản về tìm kiếm và sắp xếp trong trên kiểu dữ liệu tuyến tính mảng. Thông qua đó, trình bày một số ý tưởng và kỹ thuật cơ bản nhằm cải tiến các giải thuật.
- Chương 3: Trình bày kiểu dữ liệu con trỏ. Trên cơ sở đó, trình bày các kiểu dữ liệu động tuyến tính và có nhiều ứng dụng trong tin học là các kiểu danh sách liên kết khác nhau, ngăn xếp, hàng đợi, cũng như một số ứng dụng của chúng.
- Chương 4: Giới thiệu một loại cấu trúc dữ liệu động khác là cây và các thao tác cơ bản trên cây nhị phân, cây nhị phân tìm kiếm, cây cân bằng AVL.

Nhằm mục đích dành thời gian nhiều hơn cho sinh viên để làm các bài tập lớn, nên trong một số phần tác giả đã trình bày khá chi tiết các dạng cài đặt biến thể khác nhau cho các giải thuật. Các phần thứ yếu hoặc khá phức tạp sẽ được in cỡ chữ nhỏ dành cho sinh viên đọc thêm.

Chắn chắn rằng trong giáo trình sẽ còn nhiều khiếm khuyết, tác giả mong muốn nhận được và rất biết ơn các ý kiến quí báu đóng góp của đồng nghiệp cũng như bạn đọc để giáo trình này có thể hoàn thiện hơn nữa về mặt nội dung cũng như hình thức trong lần tái bản sau.

Đà lạt, 04/2008
Tác giả

MỤC LỤC

Chương I. GIỚI THIỆU CẤU TRÚC DỮ LIỆU, PHÂN TÍCH GIẢI THUẬT

Chương II. TÌM KIẾM VÀ SẮP XẾP TRONG

Trang

Chương III. CẤU TRÚC DANH SÁCH LIÊN KẾT

Chương I

GIỚI THIỆU CẤU TRÚC DỮ LIỆU VÀ PHÂN TÍCH GIẢI THUẬT

I.1. Quan hệ giữa cấu trúc dữ liệu và giải thuật, kiểu dữ liệu

I.1.1. Biểu diễn dữ liệu

Một mục tiêu quan trọng của tin học là nhằm giải quyết tự động những bài toán trong thế giới thực bằng máy tính điện tử. Các thông tin về bài toán cần giải quyết trên máy tính luôn được mã hoá dưới *dạng nhị phân*. Các thông tin này gồm *dữ liệu* và các *thao tác* trên các dữ liệu đó.

Việc biểu diễn dữ liệu ở dạng nhị phân rất bất tiện cho con người trong khi xử lý các bài toán, đặc biệt là các bài toán lớn và phức tạp. Chính vì lý do đó, các ngôn ngữ lập trình bậc cao đã cung cấp sẵn các cách biểu diễn *dữ liệu trừu tượng đơn giản* và *có cấu trúc*, nhằm giúp người lập trình không phải mất nhiều thời gian và công sức thực hiện thường xuyên lặp lại các thao tác sơ cấp nặng nề trên các kiểu dữ liệu nhị phân ở mức thấp. Tính trừu tượng của dữ liệu thể hiện ở chỗ nó không quá chú trọng đến những đặc điểm và ý nghĩa riêng của từng đối tượng cụ thể mà chỉ rút ra và phản ánh những tính chất chung nhất mà các đối tượng thuộc cùng một lớp có được.

I.1.2. Quan hệ giữa cấu trúc dữ liệu và giải thuật, kiểu dữ liệu

Dựa vào *bản chất chung* của từng nhóm dữ liệu, các đối tượng dữ liệu được phân thành các lớp. Mỗi *lớp dữ liệu* được thể hiện qua một kiểu dữ liệu. Một *kiểu dữ liệu T* là một tập hợp nào đó, mỗi phần tử của tập được gọi là một *thể hiện* của kiểu.

Ta đã biết *giải thuật* (hay *giải thuật) là một dãy câu lệnh rõ ràng, xác định một trình tự các thao tác trên một số đối tượng nào đó (input) sao cho sau một số hữu hạn bước thực hiện (chú ý đến tính khả thi về thời gian), ta đạt được kết quả (output) mong muốn*. *Giải thuật phản ánh các phép xử lý, còn đối tượng để xử lý bởi giải thuật chính là dữ liệu:* dữ liệu (input) đưa vào, dữ liệu trung gian và kết qủa (output) cuối cùng.

Đối với bất kỳ một lớp dữ liệu nào, nếu để ý kỹ, ta thấy trên đó luôn tồn tại những thao tác cơ bản mật thiết gắn liền với các đối tượng dữ liệu cùng kiểu đó. Khi cách biểu diễn dữ liệu thay đổi thì các thao tác gắn liền với chúng cũng thay đổi theo. Vì nếu không thì trong nhiều trường hợp việc xử lý sẽ gượng ép, thiếu tự

nhiên, khó hiểu, phức tạp không cần thiết và chương trình kém hiệu quả, lãng phí tài nguyên trên máy tính (CPU và bộ nhớ).

Chẳng hạn, đối với một chuỗi ký tự, ta có ít nhất hai cách biểu diễn chúng như được thể hiện trong ngôn ngữ lập trình Pascal và C. Với mỗi cách biểu diễn, ta sẽ có những cách xây dựng các thao tác tương ứng trên chúng khác nhau.

Một ví dụ khác, sẽ thấy rõ hơn trong các chương tiếp theo, đối với một dãy các phần tử dữ liệu cùng loại, ta có thể lưu trữ chúng ít nhất bằng hai cách: lưu bằng mảng (tĩnh, động) hay lưu trữ bằng danh sách liên kết động. Khi đó, các thao tác cơ bản trên chúng như chèn, xóa, sắp xếp sẽ thực hiện theo những cách thức khác nhau và do đó có hiệu quả khác nhau.

Do đó, khi nói đến một ***kiểu dữ liệu*** **T**, ta thường chú ý đến *hai đặc trưng* quan trọng và liên hệ mật thiết với nhau:

- *tập **V*** các *giá trị* thuộc kiểu, đó là tập các giá trị hợp lệ mà đối tượng kiểu T có thể nhận và lưu trữ;
- *tập **O*** các *phép toán (*hay *thao tác xử lý)* xác định có thể thực hiện trên các đối tượng dữ liệu kiểu đó.

Người ta thường viết: **T = <V, O>.**

Trong một ngôn ngữ lập trình cấp cao cụ thể, người ta thường xây dựng sẵn một số *kiểu dữ liệu đơn giản* hay sơ cấp xác định, chẳng hạn với C++, ta có các kiểu dữ liệu: số (nguyên, thực), ký tự, lôgic. Với kiểu số nguyên, các phép toán thường gặp là: các phép toán số học +, -, *, / (chia nguyên), % (mod, lấy phần dư) và các phép toán so sánh như: ==, !=, ≥, ≤, >, <. Với kiểu số thực, các phép toán thường gặp là: các phép toán số học +, -, *, /, và các phép toán so sánh như: ==, !=, ≥, ≤, >, <. Với kiểu lôgic, các phép toán thường gặp là: ! (not), && (and), || (or). Với kiểu ký tự, các phép toán thường gặp là: phép toán ép kiểu và các phép toán so sánh như: ==, !=, ≥, ≤, >, <, …

Dựa trên các kiểu đơn giản đã có và các *phương pháp xác định* của ngôn ngữ lập trình qui định, ta có thể xây dựng nên các *cấu trúc dữ liệu* hay *kiểu dữ liệu có cấu trúc* phức tạp hơn nhằm phản ánh tốt hơn các loại dữ liệu phong phú và đa dạng trong thế giới thực. Chẳng hạn như: kiểu mảng, kiểu cấu trúc, kiểu hợp, kiểu file, … Một trong những phép toán cơ bản trên các kiểu dữ liệu đó là: truy cập đến từng phần tử hay từng thành phần của đối tượng dữ liệu.

I.1.3. Các bước chính để giải một bài toán trên máy tính

Để giải một bài toán trên máy tính, ta thường trải qua các giai đoạn chính sau đây:

- Đặt bài toán, phân tích, đặc tả và mô hình hoá bài toán
- Chọn cấu trúc dữ liệu để biểu diễn bài toán và phát triển giải thuật (chọn *kiểu dữ liệu)*
- Mã hóa chương trình
- Thử nghiệm chương trình
- Bảo trì chương trình.

Hai giai đoạn đầu rất quan trọng, nó góp phần quyết định tính đúng đắn và hiệu quả của chương trình nhằm giải bài toán.

Vai trò của kiểu dữ liệu trong việc giải một bài toán trên máy tính

Khi đề cập đến một thao tác, cần phải xác định nó tác động lên loại đối tượng hay trên cấu trúc dữ liệu hoặc trong kiểu dữ liệu nào?

Với *mỗi mô hình dữ liệu*, có thể *có nhiều cách cài đặt bởi các cấu trúc dữ liệu khác nhau*. Trong mỗi cách cài đặt, có thể có một số phép toán được thực hiện thuận lợi, nhưng một số phép toán khác lại không thuận tiện. Khi đề cập đến một thao tác, cần phải xác định rõ nó tác động trên loại đối tượng hoặc kiểu dữ liệu nào? Khi *cấu trúc dữ liệu thay đổi* thì các *giải thuật cơ bản tương ứng với nó cũng thay đổi* theo. Vì vậy *việc chọn cấu trúc dữ liệu nào để biểu diễn mô hình sẽ phụ thuộc vào từng ứng dụng cụ thể*.

Để việc ***chọn cấu trúc dữ liệu*** biểu diễn bài toán một cách *phù hợp*, cần *chú ý* đến những *quan hệ giữa các đối tượng và thành phần dữ liệu với nhau*; ngoài ra, ta còn cần phải lưu ý đến những *phép toán cơ bản nào sẽ được thực hiện thường xuyên* trên các đối tượng dữ liệu đó. Chẳng hạn, đối với một dãy các đối tượng dữ liệu cùng loại, nếu số lượng các đối tượng này không quá lớn (để có thể lưu ở bộ nhớ trong), biến động nhiều, hơn nữa các phép toán thêm và hủy các đối tượng xảy ra rất thường xuyên thì ta nên chọn kiểu dữ liệu là danh sách liên kết động hơn là kiểu mảng tĩnh để lưu trữ dãy đối tượng này.

Khi ***xây dựng các giải thuật*** nhằm giải quyết một bài toán, ta phải *dựa trên các yêu cầu cần xử lý* để xem xét kỹ lưỡng, cũng như nên dựa trên các *đặc trưng* của bài toán và *tài nguyên (tốc độ xử lý* và *khả năng lưu trữ* của hệ thống máy tính*)* thực tế hiện có.

Tóm lại, khi *xây dựng các kiểu dữ liệu* nhằm giải quyết một bài toán cụ thể, ta nên để ý các *tiêu chuẩn* sau:

- Phản ánh đúng thực tế: có dự trù đến *khả năng biến đổi* của dữ liệu trong *chu trình sống* của nó. Đây là tiêu chuẩn rất quan trọng nhằm quyết định tính đúng đắn của toàn bộ bài toán.
- Cấu trúc dữ liệu được xây dựng cần phù hợp với các thao tác trên đó (đặc biệt là các thao tác *được sử dụng nhiều nhất*). Khi đó, việc phát triển các *giải thuật* sẽ *đơn giản, tự nhiên* hơn và đạt *hiệu quả* cao về mặt tốc độ và bộ nhớ.

- Tiết kiệm tài nguyên (tốc độ xử lý và dung lượng bộ nhớ): Đối với các giải thuật không quá tầm thường, hai yêu cầu này thường *mâu thuẫn nhau* và khó đạt được tối ưu đồng thời. Tùy theo yêu cầu của bài toán và tài nguyên thực tế, ta nên chọn giải thuật cho phù hợp.

I.2. Thiết kế và phân tích giải thuật

I.2.1. Thiết kế giải thuật theo phương pháp Top-Down

Các bài toán giải được trên máy tính ngày càng đa dạng và phức tạp. Việc xây dựng mô hình cùng với các giải thuật và cách cài đặt các chương trình giải chúng ngày càng có quy mô lớn và phức tạp, thường đòi hỏi công sức đồng thời của cả một tập thể các nhóm phân tích - thiết kế viên cũng như các thảo chương viên. Mặt khác, việc thử nghiệm, sửa chữa, bổ sung, mở rộng, bảo trì các hệ chương trình lớn chiếm tỷ lệ thời gian đáng kể so với tổng thời gian xây dựng hệ chương trình.

Để chương trình trở nên dễ hiểu, dễ kiểm tra, dễ bảo trì và dễ mở rộng hơn, đặc biệt là trong môi trường làm việc theo nhóm, người ta thường áp dụng chiến thuật ***"chia để trị"*** *bằng* ***phương pháp thiết kế từ trên xuống*** *(top-down design) hay thiết kế từ khái quát đến chi tiết.* Đó là cách phân tích bài toán, xuất phát từ dữ kiện và các mục tiêu đặt ra nhằm đưa ra các công việc chủ yếu (theo cấu trúc phân cấp, chưa vội sa đà vào tiểu tiết), rồi mới chia dần từng công việc lớn thành các công việc (*module*) chi tiết hơn; nếu các module này vẫn còn phức tạp ta lại chia tiếp chúng thành các module nhỏ hơn cho tới khi đạt đến các phần việc cơ bản mà ta đã biết cách giải quyết. Việc giải bài toán lớn ban đầu qui về việc kết hợp những lời giải của các bài toán con. Đó cũng là cơ sở của *kỹ thuật lập trình có cấu trúc*.

Khi thiết kế từng module nên chú ý đến tính *độc lập tương đối* của chúng đối với các module khác. Phương pháp thiết kế này hỗ trợ đắc lực trong việc lập trình theo nhóm của công nghệ phần mềm. Khi đó, nhiều người có thể cùng chia xẻ giải quyết các vấn đề lớn mà không cần quan tâm tới chi tiết phần việc của người khác mà sau đó vẫn có thể nối kết các module nhỏ để cả bài toán lớn được giải quyết. Quá trình này làm cho việc tìm hiểu cũng như sửa lỗi, bổ sung, mở rộng chương trình trở nên dễ dàng và đơn giản hơn.

Việc phân tích và thiết kế bài toán lớn thành các bài toán con thường chiếm thời gian lẫn công sức lớn hơn nhiều so với nhiệm vụ lập trình (coding).

I.2.2. Các chiến lược khác để thiết kế giải thuật

Ngoài chiến lược chia để trị, người ta còn dùng các phương pháp thiết kế giải thuật sau: phương pháp tham lam, phương pháp qui hoạch động, phương pháp quay lui, phương pháp nhánh và cận.

Phương pháp *tham lam* thường dùng để tìm nghiệm tối ưu trong một tập nghiệm chấp nhận được *S* nào đó được xây dựng theo một hàm chọn để bổ sung những phần tử vào *S* theo một cách thích hợp.

Phương pháp *qui hoạch động* sử dụng kỹ thuật *"đi từ dưới lên"*: xuất phát từ nghiệm của những bài toán con sơ cấp (được lưu giữ trong một bảng nhằm tránh mất công sức giải lại những bài toán con này sẽ phát sinh khi cần giải những bài con lớn hơn sau này), ta xây dựng nghiệm của những bài toán con lớn hơn và lưu tiếp vào bảng; cứ tiếp tục như vậy cho đến khi tìm được nghiệm của bài toán lớn ban đầu từ bảng.

Phương pháp *quay lui* thường dùng để *tìm một* hoặc *tất cả nghiệm* của bài toán dưới dạng một vectơ nghiệm có thể chưa biết trước độ dài của nó và có thể được xác định dần trong quá trình giải. Đây là một kỹ thuật rất quan trọng trong việc thiết kế giải thuật.

Phương pháp *nhánh và cận* là một dạng *cải tiến của phương pháp quay lui* để *tìm nghiệm tối ưu* của bài toán. Trong quá trình từng bước mở rộng nghiệm từng phần để đạt đến nghiệm tối ưu của bài toán (dưới dạng vectơ), nếu biết các nghiệm mở rộng đều có hàm giá lớn hơn giá của nghiệm tốt nhất ở thời điểm đó, thì ta không cần mở rộng nghiệm một phần theo nhánh này nữa và quay lui sang tìm nghiệm trên nhánh khác có triển vọng hơn.

Các chiến lược này sẽ được nghiên cứu chi tiết trong các học phần tiếp theo.

I.2.3. Phân tích giải thuật và độ phức tạp của giải thuật

a. Các vấn đề cần lưu ý khi phân tích giải thuật

- *Tính đúng đắn* của giải thuật: cần trả lời câu hỏi liệu giải thuật có thể hiện đúng lời giải của bài toán hay không? Thông thường người ta cài đặt giải thuật đó trên máy tính và thử nghiệm nó với một số bộ dữ liệu mẫu nào đó rồi so sánh kết quả thử nghiệm với kết quả được lấy từ những thông tin và phương pháp khác mà ta đã biết chắc đúng. Nhưng *cách thử này chỉ phát hiện được tính sai* chứ chưa thể bảo đảm được tính đúng của giải thuật. Để *chứng minh được tính đúng đắn* của giải thuật nhiều khi đòi hỏi phải sử dụng các công cụ toán học khá phức tạp, nhưng đây là một công việc không phải luôn luôn dễ dàng.

- *Tính đơn giản* của giải thuật: thể hiện qua tính *dễ hiểu, tự nhiên, dễ lập trình, dễ chỉnh lý*. Thông thường các giải thuật quá đơn sơ chưa hẳn là cách tốt nhất và nó thường gây tốn phí thời gian và bộ nhớ khi thực hiện. Nhưng trên thực tế ta nên *cân nhắc* giữa tính đơn giản của giải thuật và thời gian lẫn công sức để xây dựng các giải thuật tinh tế, hiệu quả hơn nhưng chỉ sử dụng quá ít lần với bộ dữ liệu quá nhỏ với điều kiện thời gian hạn chế trong một môi trường lập trình thực tế.

- *Tốc độ thực hiện và dung lượng bộ nhớ cần chiếm dụng của giải thuật*: Thông thường *hiếm khi cả hai yêu cầu tối ưu về thời gian và bộ nhớ được thỏa mãn đồng thời*. Các giải thuật không tầm thường nếu có tốc độ thực hiện cao thì

thường chiếm bộ nhớ nhiều và ngược lại. Ở đây ta hạn chế chỉ xét yêu cầu về thời gian thực hiện của giải thuật.

b. Độ phức tạp của giải thuật

- Thời gian thực hiện một giải thuật phụ thuộc vào khá nhiều yếu tố:

- *Kích thước dữ liệu n đưa vào*: ta gọi thời gian thực hiện của giải thuật trên bộ dữ liệu này là một hàm của n : *T(n)*

- Các kiểu lệnh và tốc độ xử lý của máy tính, ngôn ngữ lập trình và chương trình dịch ngôn ngữ ấy. Nhưng các loại yếu tố này *phụ thuộc* vào cách *cài đặt* và loại *máy tính* trên đó giải thuật được cài đặt. Vì vậy khi xây dựng *T(n)* không nên dựa vào chúng.

- Khi xây dựng hàm T(n) cho một giải thuật người ta *thường chỉ xét các thao tác đặc trưng cho giải thuật đó (thời gian thực hiện các thao tác này nhiều hơn đáng kể so với thời gian thực hiện các loại thao tác khác)*. Chẳng hạn, khi xét các giải thuật sắp xếp *n* mục dữ liệu với cấu trúc "lưu trữ trong" ta thường chú ý tới số lần đổi chỗ và so sánh các mục dữ liệu theo một trường khoá nào đó.

- *Tình trạng của dữ liệu*: Thời gian thực hiện giải thuật không chỉ phụ thuộc vào kích thước *n* của dữ liệu mà *còn phụ thuộc vào chính tình trạng của dữ liệu đó*. Chẳng hạn, số các thao tác cơ bản để sắp xếp theo thứ tự tăng một dãy số đưa vào đã có đúng thứ tự sẽ khác nhiều so với dãy chưa được sắp hay đã sắp theo thứ tự ngược lại. Vì vậy, khi xét độ phức tạp T(n) của giải thuật ta *thường xét các trường hợp: thuận lợi nhất, xấu nhất và trung bình (thường khó xét vì trong nhiều trường hợp đòi hỏi các công cụ toán học phức tạp).*

Cách đánh giá thời gian thực hiện giải thuật độc lập với máy tính và chỉ phụ thuộc vào bản thân giải thuật và dữ liệu như vậy sẽ dẫn tới khái niệm "***độ phức tạp của giải thuật***" hay cấp độ lớn của thời gian thực hiện giải thuật.

- *Gọi T(n) là độ phức tạp của một giải thuật, nếu tồn tại: một hàm g(n) không âm, các hằng số dương C và n_0 sao cho:*

  $T(n) \le C\,g(n)$ *khi* $n \ge n_0$ *(1)*

Khi đó ta nói: *T(n) có cấp g(n)* và viết: ***T(n) = O(g(n))***.

+ *Lưu ý:*

- Ta nên chọn *cận trên g(n)* có "cấp nhỏ nhất" thỏa mãn tính chất (1).
- *T(n) có cấp g(n) nếu :*

$$\lim \frac{T(n)}{g(n)} = C > 0, \quad n \to \infty$$

- Thông thường ta dùng các hàm sau để đánh giá độ phức tạp của giải thuật: $1 << \log_2 n << n << n \log_2 n << n^2 << \ldots << n^k$ *(k>= 2, độ phức tạp loại đa thức)* $<<$ *(độ phức tạp loại mũ)* $2^n << n! << n^n$

trong đó, ký hiệu : f(n) << g(n) có nghĩa là "f(n) nhỏ hơn g(n) rất nhiều" khi n đủ lớn hay:

$$\lim \frac{f(n)}{g(n)} = 0, \quad n \to \infty$$

Bảng sau đây cho ta hình dung về độ tăng nhanh của các lớp giải thuật có độ phức tạp đa thức và mũ theo số lượng *n* các mục dữ liệu đầu vào. Giả sử ta cài đặt các giải thuật trên một máy tính với tốc độ xử lý 1 tỉ phép tính trong 1 giây (s).

N	*$Log_2(n)$ (s)*	*n (s)*	*$n*Log_2(n)$ (s)*	*n*n (s)*	***2^n(năm)***	***n! (năm)***	***n^n (năm)***
10	3 e-09	1 e-08	3 e-08	1 e-07	*3 e-14*	*1 e-10*	*3 e-07*
50	6 e-09	5 e-08	3 e-07	3 e-06	*4 e-02*	***1 e+48***	***3 e+68***
100	7 e-09	1 e-07	7 e-07	1 e-05	***4 e+13***	***3 e+141***	***3 e+183***

c Một số quy tắc để xác định độ phức tạp của giải thuật

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai đoạn chương trình P_1 và P_2 mà $T_1(n) = O(f(n))$ và $T_2(n) = O(g(n))$.

- <u>Quy tắc tổng</u>: T*hời gian thực hiện liên tiếp P_1 và P_2* là: $T_1(n) + T_2(n) = O(max(f(n),g(n)))$.

<u>Ví dụ</u>: nếu $f(n) \le g(n), \ \forall n \ge n_0$ thì $O(f(n) + g(n)) = O(g(n))$

- <u>Quy tắc nhân</u>: *Thời gian thực hiện P_1 và P_2 lồng nhau* là: $T_1(n)\ T_2(n) = O(f(n).g(n))$.

<u>Ví dụ</u>: P_1 là một vòng lặp, P_2 là một thao tác trong P_1.

d. Các bước phân tích giải thuật

- Xác định *đặc trưng dữ liệu* được dùng làm dữ liệu *nhập* và quyết định sự phân tích nào là phù hợp.
- Xác định các *thao tác cơ bản trừu tượng* của giải thuật để *tách biệt sự phân tích với sự cài đặt.*
- *Phân tích về mặt toán học độ phức tạp* của giải thuật trong các *trường hợp: tốt nhất, xấu nhất và trung bình*. Để đánh giá độ phức tạp của giải thuật trong trường hợp trung bình thường đòi hỏi những công cụ toán học khá tinh vi và khó; vì vậy trong nhiều trường hợp, ta thường hạn chế trên những đánh giá ước lượng chặn trên và tránh sa đà vào các tiểu tiết phức tạp.

* *Ví dụ*: Xét giải thuật tìm xem một phần tử X có mặt trong một vector có *n* phần tử V = {v_1,v_2, .., v_n} cho trước hay không?

Boolean TìmKiếm(ptu X, ptu V[], int n)
Bước 1: Thấy = False;
Thứ = 1;
Bước 2: Trong khi (not(Thấy) and Thứ ≤ n)
{ if ($v_{Thứ}$ == X) Thấy = True;
else Thứ = Thứ + 1;
}
Bước 3: Trả về trị Thấy;

Phép toán cơ bản trong giải thuật tìm kiếm trên là *phép so sánh khóa dữ liệu* $v_{Thứ}$ với X.

- Trường hợp tốt nhất xảy ra khi X bằng v_1:
$T_{tốt}(n) = O(1)$.

- Trường hợp xấu nhất xảy ra khi X chỉ bằng v_n hoặc không tìm thấy:
$T_{xấu}(n) = O(n)$.

- Trường hợp trung bình: Gọi q là xác suất để X rơi vào một phần tử nào đó của V và giả sử X có *phân bố đều* trên n phần tử phân biệt của V thì xác suất để X rơi vào phần tử v_i là: $p_i = q/n$; còn xác suất để X không rơi vào phần tử nào của V sẽ là: 1 - q.

Độ phức tạp trung bình của giải thuật là:

$$T_{tb}(n) = \sum_{i=1}^{n} p_i.i + (1-q)n$$

$$T_{tb}(n) = q\sum_{i=1}^{n} i/n + (1-q)n$$

$$= q(n+1)/2 + (1-q)n$$

$$= n(1-q/2) + q/2$$

Nếu q=1 (nghĩa là luôn tìm thấy X trong V) thì : $T_{tb}(n) = (n+1)/2$
Nếu q=1/2 (nghĩa là khả năng tìm thấy và không tìm thấy X trong V bằng nhau) thì : $T_{tb}(n) = (3n+1)/4$
Nếu q= 0 (nghĩa là không tìm thấy X trong V) thì : $T_{tb}(n) = n$
Tóm lại: $T_{tb}(n) = O(n)$.

I.2.4. Qui ước về ngôn ngữ mã giả

Để tiện cho việc thực hành cho học viên (trên ngôn ngữ lập trình C hay C++), trong giáo trình sẽ sử dụng *ngôn ngữ mã giả tựa ngôn ngữ C++* (thật ra nó chỉ khác ngôn ngữ mã giả tựa Pascal không đáng kể) để mô tả cấu trúc dữ liệu và các cấu trúc điều khiển trong các giải thuật.

- Lệnh ghép: dãy lệnh nằm giữa cặp dấu ngoặc kép { … }
- Cấu trúc điều khiển: "nếu (điều kiện đúng) thì thực hiện lệnh S":

```
    if (ĐiềuKiện) S;
```

hoặc:

```
    if (ĐiềuKiện) S1;
    else S2;
```

- Cấu trúc điều khiển nhiều chọn lựa:

```
    switch (BiểuThứcVôHướng)
    { case  Trị_1: S1; break;
      case  Trị_2: S2; break;
            …
      case  Trị_n: Sn; break;
     [default : S;]
    };
```

- Cấu trúc lặp:

```
    for (LệnhKhởiĐầu; ĐiềuKiệnLặp; LệnhThayĐổiĐiềuKiệnLặp) S;
    while (ĐiềuKiện) S;
    do S while (ĐiềuKiện);
    repeat S until (ĐiềuKiện);
```

- Phép gán: =
- Phép toán lôgic: && (and), || (or), ! (not) và trị lôgic kiểu boolean: True, False.
- Quan hệ so sánh: ==, !=, >, <, ≤, ≥
- Khai báo chương trình con viết dưới dạng hàm:

KiểuTrảVềCủaHàm **TênHàm**(KiểuThamTrị ThamTrị, KiểuThamChiếu **&**ThamChiếu)

Chương II

TÌM KIẾM VÀ SẮP XẾP TRONG

II.1. Giới thiệu về sắp xếp và tìm kiếm

II.1.1. Sắp xếp

a. Định nghĩa sắp xếp

Cho dãy *X* gồm *n* phần tử $x_1, x_2,..., x_n$ có cùng một kiểu dữ liệu T_0. *Sắp thứ tự n* phần tử này là một *hoán vị các phần tử thành dãy* $x_{k1}, x_{k2},..., x_{kn}$ sao cho với một *hàm thứ tự f cho trước, ta có :*

$$f(x_{k1}) \propto f(x_{k2}) \propto ... \propto f(x_{kn}).$$

trong đó: $\propto$ *là một quan hệ thứ tự.* Ta thường gặp $\propto$ là quan hệ thứ tự "$\leq$" thông thường.

b. ***Phân loại phương pháp sắp xếp***

Dựa trên *tiêu chuẩn lưu trữ* dữ liệu ở bộ nhớ trong hay ngoài mà ta chia các phương pháp sắp xếp thành hai loại:

* *Sắp xếp trong*: Với các phương pháp sắp xếp trong, toàn bộ dữ liệu được đưa vào bộ nhớ trong (bộ nhớ chính). Đặc điểm của phương pháp sắp xếp trong là khối lượng dữ liệu bị hạn chế nhưng bù lại, thời gian sắp xếp lại nhanh.

* *Sắp xếp ngoài*: Với các phương pháp sắp xếp ngoài, toàn bộ dữ liệu được lưu ở bộ nhớ ngoài. Trong quá trình sắp xếp, chỉ một phần dữ liệu được đưa vào bộ nhớ chính, phần còn lại nằm trên thiết bị trữ tin. Đặc điểm của loại sắp xếp ngoài là khối lượng dữ liệu ít bị hạn chế, nhưng thời gian sắp xếp lại chậm (do thời gian chuyển dữ liệu từ bộ nhớ phụ vào bộ nhớ chính để xử lý và kết quả xử lý được đưa trở lại bộ nhớ phụ thường khá lớn).

c. Vài qui ước về kiểu dữ liệu khi xét các thuật toán sắp xếp

Thông thường, T_0 có kiểu cấu trúc gồm *m* trường thành phần $T_1, T_2, ..., T_m$. Hàm thứ tự *f* là một ánh xạ từ miền trị của kiểu T_0 vào miền trị của một số thành phần $\{T_{ik}\}_{1\leq ik \leq p}$, trên đó có một quan hệ thứ tự α.

Không mất tính tổng quát, ta có thể giả sử *f* là ánh xạ từ miền trị của T_0 vào *miền trị* của *một thành phần dữ liệu* đặc biệt (mà ta gọi là *khóa- key*) , trên đó có một quan hệ thứ tự α.

Khi đó, kiểu dữ liệu chung T_0 của các phần tử x_i thường được cài đặt bởi cấu trúc:

```
typedef struct  { KeyType  key;
                  DataType  Data;
                } ElementType;
```

Khi đó bài toán đưa về sắp xếp dãy $\{x_i.key\}_{1\leq i\leq n}$.

Để đơn giản trong trình bày, ta có thể giả sử T_0 *chỉ gồm* trường khóa, α là quan hệ thứ tự $\leq$ thông thường và f là hàm đồng nhất và ta chỉ cần xét các phương pháp sắp xếp tăng trên dãy đơn giản $\{x_i\}_{1\leq i\leq n}$. Trong chương này, khi xét các phương pháp sắp xếp trong, dãy ***x*** thường được lưu trong mảng tĩnh như sau:

```
#define MAX_SIZE …
// Kích thước tối đa của mảng cần sắp theo thứ tự tăng
typedef  ....  ElementType; // Kiểu dữ liệu chung cho các phần tử của mảng
typedef  ElementType  mang[MAX_SIZE] ;  // Kiểu mảng
mang  x;
```

Trong phần cài đặt các thuật toán sắp xếp sau này, ta thường sử dụng các phép toán: đổi chỗ *HoánVị*(x,y), gán *Gán*(x,y), so sánh *SoSánh*(x,y) như sau:

```
void HoánVị(ElementType &x, ElementType &y)
{ ElementType tam;
    Gán(tam,  x);
    Gán(x, y);
    Gán(y, tam);
    return ;
}

void Gán(ElementType &x, ElementType y)
{
    // Gán y vào x, tùy từng kiểu dữ liệu mà ta có phép gán cho hợp lệ
    return;
}

int SoSánh(ElementType x, ElementType y)
{
    // Hàm trả về trị: 1 nếu x > y
    //                 0 nếu x == y
    //                -1  nếu x < y
    // tùy theo kiểu ElementType mà ta dùng các quan hệ <, >, == cho hợp lệ
}
```

Khi đánh giá *độ phức tạp của mỗi thuật toán sắp xếp,* ta thường *chỉ tính số lần so sánh khóa (SS), số lần hoán vị khóa (HV)* hoặc *số lần Gán (G)* trong thuật toán đó.

II.1.2. Tìm kiếm

a. Định nghĩa tìm kiếm

Cho trước một phần tử *Item* và dãy *X* gồm *n* phần tử $x_1, x_2,..., x_n$ đều có cùng kiểu T_0. Bài toán tìm kiếm là *xem Item có mặt trong dãy X hay không*? (hay tổng quát hơn: xem trong dãy *X* có phần tử nào thỏa mãn một tính chất *TC* cho trước nào đó liên quan đến *Item* hay không?)

b. Phân loại các phương pháp tìm kiếm

Cũng tương tự như sắp xếp, ta cũng có 2 loại phương pháp *tìm kiếm trong và ngoài* tùy theo dữ liệu được lưu trữ ở bộ nhớ trong hay ngoài.

Với từng nhóm phương pháp, ta lại phân biệt các phương pháp tìm kiếm tùy theo dữ liệu ban đầu đã được sắp hay chưa. Chẳng hạn đối với trường hợp dữ liệu đã được sắp và lưu ở bộ nhớ trong, ta có 2 phương pháp tìm kiếm: tuyến tính hay nhị phân.

Khi cài đặt các thuật toán tìm kiếm, ta cũng có các qui ước tương tự cho kiểu dữ liệu và các phép toán cơ bản trên kiểu đó như đối với các phương pháp sắp xếp đã trình bày ở trên.

Trong chương này, ta chỉ hạn chế xét các phương pháp tìm kiếm và sắp xếp trong.

II.2. Phương pháp tìm kiếm trong

Bài toán:

Input : - dãy $X = \{x_1, x_2,..., x_n\}$ gồm n mục dữ liệu

- Item: mục dữ liệu cần tìm cùng kiểu dữ liệu với các phần tử của *X*

Output: Trả về:

- trị 0, nếu không thấy Item trong X
- vị trí đầu tiên *i* $(1 \le i \le n)$ trong X sao cho $x_i \equiv$ Item.

II.2.1. Phương pháp tìm kiếm tuyến tính

a. Dãy chưa được sắp

Đối với dãy bất kỳ chưa được sắp thứ tự, thuật toán tìm kiếm đơn giản nhất là *tìm tuần tự từ đầu đến cuối dãy.*

- *Thuật toán*

int TìmTuyếnTính(x, n, Item)

- Bước 1: VịTrí = 1;
- Bước 2: if ((VịTrí ≤ n) and ($x_{VịTrí}$!= Item))
  { VịTrí = VịTrí + 1;
  Quay lại đầu bước 2;
  }
  else chuyển sang bước 3;
- Bước 3: if (VịTrí > n) VịTrí = 0; //không thấy
  Trả về trị VịTrí;

- *Cài đặt*

```
int TìmTuyếnTính (mang x, int n, ElementType Item)
{ int VịTrí = 0;
    while ((VịTrí < n) && (x[VịTrí] != Item))
        VịTrí = VịTrí + 1 ;
    if (VịTrí ≥ n) VịTrí = 0;     //không thấy
    else VịTrí++;
    return(VịTrí);
}
```

* *Chú ý*: Để cài đặt thuật toán trên (cũng tương tự như thế với các thuật toán tiếp theo) với danh sách tuyến tính nói chung thay cho cách cài đặt danh sách bằng mảng, ta chỉ cần thay các câu lệnh hay biểu thức sau:

VịTrí = 1; VịTrí = VịTrí + 1; (VịTrí ≤ n) ; $x_{VịTrí}$;

trong thuật toán tương ứng bởi:

ĐịaChỉ = ĐịaChỉ phần tử (dữ liệu) đầu tiên; ĐịaChỉ = ĐịaChỉ phần tử kế tiếp;
(ĐịaChỉ != ĐịaChỉ kết thúc); Dữ liệu của phần tử tại ĐịaChỉ;

**** Độ phức tạp*** *của thuật toán tìm kiếm tuyến tính* (trên dãy chưa được sắp) trong trường hợp:

- *tốt nhất* (khi Item ≡ x_1): $T_{tốt}(n) = O(1)$
- *tồi nhất* (khi không có Item trong dãy hoặc Item chỉ trùng với x_n)*:*
  $T_{xấu}(n) = O(n)$
- *trung bình:* $T_{tbình}(n) = O(n)$

**** Thuật toán tìm kiếm tuyến tính cải tiến bằng kỹ thuật lính canh***

Để *giảm bớt phép so sánh chỉ số* trong biểu thức điều kiện của lệnh *if* hay *while* trong thuật toán trên, ta dùng thêm một biến phụ đóng vai trò *lính canh bên phải* (hay trái) x_{n+1} = Item (hay x_0 = Item).

- *Thuật toán*

int TìmTuyếnTính_CóLínhCanh(x, n, Item)

- Bước 1: VịTrí = 1; x_{n+1} = Item; // phần tử cầm canh
- Bước 2: if ($x_{VịTrí}$!= Item)
  { VịTrí = VịTrí + 1;
  Quay lại đầu bước 2;
  }
  else chuyển sang bước 3;
- Bước 3: if (VịTrí == n+1) VịTrí = 0; // thấy giả hay không thấy !
  Trả về trị VịTrí;

- *Cài đặt*

int TìmTuyếnTính_CóLínhCanh(mang x, int n, ElementType Item)

```
{ int VịTrí = 0;
        x[n] =  Item;                   // phần tử cầm canh
        while (x[VịTrí] != Item) VịTrí = VịTrí + 1;
        if (VịTrí == n) VịTrí = 0;   // thấy giả hay không thấy !
        else VịTrí++;
        return(VịTrí);
}
```

b. Dãy đã được sắp

Đối với dãy đã được sắp thứ tự (không mất tính tổng quát, ta có thể giả sử *tăng dần*), ta có thể cải tiến thuật toán tìm kiếm tuyến tính có lính canh như sau: ta sẽ dừng việc tìm kiếm khi tìm thấy hoặc tại thời điểm *i* đầu tiên gặp phần tử x_i mà: $x_i \geq$ Item.

- *Thuật toán*

int TìmTuyếnTính_TrongMảngĐãSắp_CóLínhCanh(a, Item, n)

- Bước 1: VịTrí = 1; x_{n+1} = Item; // phần tử cầm canh
- Bước 2: if ($x_{VịTrí}$ < Item)
  { VịTrí = VịTrí + 1;
  Quay lại đầu bước 2;
  }
  else chuyển sang bước 3;
- Bước 3: if ((VịTrí == n+1) or (VịTrí < n+1 and $x_{VịTrí}$ > Item))
  VịTrí = 0; // thấy giả hoặc không thấy !
  Trả về trị VịTrí;

- *Cài đặt*

int TìmTuyếnTính_TrongMảngĐãSắp_CóLínhCanh (mang x, ElementType Item, int n)

```
{ int VịTrí = 0;
        x[n] =  Item;                   // phần tử cầm canh
        while (x[VịTrí] < Item) VịTrí = VịTrí + 1;
        if (VịTrí < n  &&  (x[VịTrí] == Item))  VịTrí++;
        else VịTrí = 0;// thấy giả hoặc không thấy !
        return(VịTrí);
```

}

* Tuy có tốt hơn phương pháp tìm kiếm tuyến tính trong trường hợp mảng chưa được sắp, nhưng trong trường hợp này thì độ phức tạp trung bình vẫn có cấp là *n*:

$$T_{tbình} = O(n)$$

Đối với mảng đã được sắp, để giảm hẳn độ phức tạp trong trường hợp trung bình và kể cả trường hợp xấu nhất, ta sử dụng ý tưởng "chia đôi" thể hiện qua phương pháp tìm kiếm nhị phân sau đây.

II.2.2. Phương pháp tìm kiếm nhị phân.

Ý tưởng của phương pháp: Trước tiên, so sánh *Item* với phần tử đứng giữa dãy $x_{giữa}$, nếu thấy (*Item* = $x_{giữa}$) thì dừng; ngược lại, nếu *Item* < $x_{giữa}$ thì ta sẽ tìm Item trong dãy con trái: $x_1, \ldots, x_{giữa-1}$, nếu không ta sẽ tìm *Item* trong dãy con phải: $x_{giữa+1}, \ldots, x_n$. Ta sẽ thể hiện ý tưởng trên thông qua thuật toán lặp sau đây.

- *Thuật toán*

int TìmNhịPhân(x, Item, n)

- Bước 1: ChỉSốĐầu = 1; ChỉSốCuối = n;
- Bước 2: if (ChỉSốĐầu <= ChỉSốCuối)
  { ChỉSốGiữa = (ChỉSốĐầu + ChỉSốCuối)/2; *// lấy thương nguyên*
  if (Item == $x_{ChỉSốGiữa}$) Chuyển sang bước 3;
  else { if (Item < $x_{ChỉSốGiữa}$) ChỉSốCuối = ChỉSốGiữa -1;
  else ChỉSốĐầu = ChỉSốGiữa +1;
  Quay lại đầu bước 2; *// Tìm tiếp trong nửa dãy con còn lại*
  }
  }
- Bước 3: if (ChỉSốĐầu <= ChỉSốCuối) return (ChỉSốGiữa);
  else return (0); // Không thấy

- *Cài đặt*

```
int TimNhiPhan(mang x, ElementType Item, int n)
{ int  Đầu = 0, Cuối = n-1;
      while  (Đầu ≤ Cuối)
      {   Giữa = (Đầu + Cuối)/2;
            if   (Item ==  x[Giữa]) break;
            else     if   (Item <  x[Giữa]) Cuối = Giữa -1
                     else  Đầu =  Giữa + 1;
      }
      if  (Đầu ≤ Cuối) return (Giữa+1);
      else return (0);
```

}

Dựa trên ý tưởng đệ qui của thuật toán, ta cũng có thể viết lại thuật toán trên dưới dạng đệ qui, tất nhiên khi đó sẽ lãng phí bộ nhớ hơn ! Tại sao ? (xem như bài tập).

- *Độ phức tạp của thuật toán trong trường hợp trung bình và xấu nhất*:

$$\mathbf{T_{tbình}(n) = T_{xấu}(n) = O(\log_2 n)}$$

Do đó đối với *dãy được sắp*, phương pháp *tìm kiếm nhị phân* sẽ *hiệu quả hơn nhiều* so với phép tìm kiếm tuyến tính, đặc biệt khi *n* lớn.

II.3. Phương pháp sắp xếp trong

Có 3 nhóm chính các thuật toán sắp xếp trong (đơn giản và cải tiến):

* *Phương pháp sắp xếp chọn (Selection Sort):* Trong nhóm các phương pháp này, tại mỗi bước, dùng các phép so sánh, ta *chọn phần tử cực trị toàn cục (nhỏ nhất* hay *lớn nhất)* rồi *đặt nó vào đúng vị trí mút* tương ứng của dãy con còn lại chưa sắp (phương pháp *chọn trực tiếp*). Trong quá trình chọn, có thể *xáo trộn các phần tử ở các khoảng cách xa nhau một cách hợp lý (sao cho những thông tin đang tạo ra ở bước hiện tại có thể có ích hơn cho các bước sau)* thì sẽ được phương pháp sắp chọn cải tiến *HeapSort*.

* *Phương pháp sắp xếp đổi chỗ (Exchange Sort):* Thay vì chọn trực tiếp phần tử cực trị của các dãy con, trong phương pháp sắp xếp đổi chỗ, ở mỗi bước ta *dùng các phép hoán vị liên tiếp trên các cặp phần tử kề nhau không đúng thứ tự để xuất hiện các phần tử này ở mút* của các dãy con còn lại cần sắp (phương pháp *nổi bọt BubbleSort, ShakeSort*). Nếu cũng sử dụng các phép *hoán vị* nhưng *trên các cặp phần tử không nhất thiết luôn ở kề nhau* một cách hợp lý thì ta *định vị đúng được các phần tử* (*không nhất thiết phải luôn ở mép* các dãy con cần sắp) và sẽ thu được phương pháp *QuickSort rất hiệu quả*.

* *Phương pháp sắp xếp chèn (Insertion Sort)*: Theo cách *tiếp cận từ dưới lên (Down-Top),* trong phương pháp *chèn trực tiếp*, tại mỗi bước, *xuất phát từ dãy con liên tục đã được sắp, ta tìm vị trí thích hợp để chèn vào dãy* con đó một *phần tử mới để thu được một dãy con mới dài hơn vẫn được sắp (*phương pháp *chèn trực tiếp)*. Thay vì chọn các dãy con liên tục được sắp dài hơn, nếu ta *chọn các dãy con ở các vị trí cách xa nhau theo* một *qui luật khoảng cách giảm dần* hợp lý thì sẽ thu được phương pháp sắp chèn cải tiến *ShellSort*.

II.3.1. <u>Phương pháp sắp xếp chọn đơn giản</u>

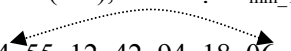
a. Ý tưởng phương pháp

Với mỗi bước lặp thứ i (i = 1, ..., n-1) chọn trực tiếp phần tử nhỏ nhất x_{min_i} trong từng dãy con có thể chưa được sắp $x_i, x_{i+1}, ..., x_n$ và đổi chỗ phần tử x_{min_i} với phần tử x_i. Cuối cùng, ta được dãy sắp thứ tự $x_1, x_2, ..., x_n$.

<u>*Ví dụ*</u>: Sắp xếp tăng dãy:

44, 55, 12, 42, 94, 18, 06, 67

Ở bước thứ 1 (i=1), tìm được $x_{min\_1} = x_7 = 6$, đổi chỗ, x_{min_1} với x_1:



44, 55, 12, 42, 94, 18, 06, 67

Kết qủa sau mỗi bước lặp:

i = 1 :	**06**	55	12	42	94	18	*44*	67
i = 2 :	06	**12**	*55*	42	94	18	44	67
i = 3 :	06	12	**18**	42	94	*55*	44	67
i = 4 :	06	12	18	***42***	94	55	44	67
i = 5 :	06	12	18	42	**44**	55	*94*	67
i = 6 :	06	12	18	42	44	***55***	94	67
i = 7 :	06	12	18	42	44	55	**67**	***94***

b. Thuật toán

SắpXếpChọn(x, n)

- Bước 1: i = 1;
- Bước 2: Tìm phần tử $x_{ChiSoMin}$ nhỏ nhất trong dãy $x_i, x_{i+1}, ..., x_n$
  Hoán Vị x_i và $x_{ChiSoMin}$;
  // Chuyển phần tử nhỏ nhất vào vị trí của x_i
- Bước 3: if (i < n)
  { i = i+1;
  Quay lại đầu bước 2;
  }
  else Dừng;

c. Cài đặt

```
void SắpXếpChọn(mang x, int n)
{ int ChiSoMin;
    for (int  i = 0; i < n -1 ;  i++)
    { ChiSoMin = i;
      for (int j = i + 1; j < n; j++)
          if (x[j] < x[ChiSoMin]) ChiSoMin = j;
      if (ChiSoMin > i)   HoánVị(x[i],x[ChiSoMin]);
    }
    return;
}
```

d. Độ phức tạp thuật toán

+ Do, trong mọi trường hợp, ở bước thứ *i (∀i = 1, ..., n-1)* luôn cần *n-i* phép so sánh khóa nên:

$$SS_{xấu} = SS_{tốt} = \sum_{i=1}^{n-1} (n\text{-}i) = \frac{n(n-1)}{2}$$

+ Trong trường hợp xấu nhất (khi dãy đã được sắp theo thứ tự ngược lại), ở bước thứ *i* ta phải đổi chỗ khóa *1* lần :

$$HV_{xấu} = \sum_{i=1}^{n-1} 1 = n - 1$$

+ Trong trường hợp tốt nhất (khi dãy đã được sắp), ở bước thứ *i* ta không phải đổi chỗ khóa lần nào:

$$HV_{tốt} = \sum_{i=1}^{n-1} 0 = 0$$

Tóm lại, độ phức tạp thuật toán:

$$T(n) = T_{tốt}(n) = T_{xấu}(n) = O(n^2).$$

II.3.2. Phương pháp sắp xếp chèn đơn giản

a. Ý tưởng phương pháp:

Giả sử dãy $x_1, x_2, ..., \mathbf{x_{i-1}}$ *đã được sắp thứ tự*. Khi đó, tìm vị trí thích hợp để chèn x_i vào dãy $x_1, x_2, ..., x_{i-1}$, sao cho dãy mới dài hơn một phần tử $x_1, x_2, ..., \mathbf{x_{i-1}, x_i}$ vẫn được sắp thứ tự. Thực hiện cách làm trên lần lượt với mỗi $i = 2, 3, ..., n$, ta sẽ thu được dãy có thứ tự.

Ví dụ : Sắp xếp dãy

67, 33, 21, 84, 49, 50, 75.

Kết qủa sau mỗi bước lặp:

i=2	**33**	67	21	84	49	50	75
i=3	**21**	33	67	84	49	50	75
i=4	21	33	67	**84**	49	50	75
i=5	21	33	**49**	67	84	50	75
i=6	21	33	49	**50**	67	84	75
i=7	21	33	49	50	67	**75**	84

b. Nội dung thuật toán

Để tăng tốc độ tìm kiếm (bằng cách giảm số biểu thức so sánh trong điều kiện lặp), ta dùng thêm *lính canh bên trái* $x_0 = x_i$ trong việc tìm vị trí thích hợp để chèn x_i vào dãy đã sắp thứ tự $x_1, x_2, ..., x_{i-1}$ để được một dãy mới vẫn tăng $x_1, x_2, ..., x_{i-1}, x_i$, (với $i = 2,..., n$).

***SắpXếpChèn*(x, n)**

- Bước 1: i = 2; // xuất phát từ dãy $x_1, x_2, ..., \mathbf{x_{i-1}}$ *đã được sắp*
- Bước 2: $x_0 = x_i$; // lưu x_i vào x_0 - đóng vai trò lính canh trái
  Tìm vị trí *j* thích hợp trong dãy $x_1, x_2, ..., \mathbf{x_{i-1}}$ để chèn x_i vào;
  //vị trí *j* đầu tiên từ phải qua trái bắt đầu từ $\mathbf{x_{i-1}}$ sao cho $x_j \le x_0$
- -Bước 3: Dời chỗ các phần tử $x_{j+1}, ..., \mathbf{x_{i-1}}$ sang phải một vị trí;
  if (j < i-1) $x_{j+1} = x_0$;
- -Bước 4: if (i < n)
  { i = i+1;
  Quay lại đầu bước 2;
  }
  else Dừng;

c. Cài đặt thuật toán

Áp dụng một mẹo nhỏ, có thể áp dụng (một cách máy móc !) ý tưởng trên để cài đặt thuật toán trong C (*bài tập*). Lưu ý rằng trong C hay C++, với *n* phần tử của mảng x[i], *i* được đánh số bắt đầu từ *0 tới n -1;* do đó, để cài đặt thuật toán này, thay cho lính canh trái như trình bày ở trên, ta sẽ dùng *lính canh bên phải* x_{n+1} (≡ x[n]) và chèn x_i thích hợp vào dãy đã sắp tăng $x_{i+1}, ..., x_n$ để được một dãy mới vẫn tăng $x_i, x_{i+1}, ..., x_n$, với mọi $i = n-1, ..., 1$.

```
void SắpXếpChèn(mang x, int n)
{
        for ( int  i = n -2 ; i >= 0 ;  i--)
        { x[n] =  x[i];                    // lính canh phải
          j =  i+1;
          while (x[ j ] < x[n])
          { x[ j-1] =  x[ j ];             // dời x[ j] qua trái một vị trí
            j++;
          }
          if (j > i+1) x[ j-1] = x[n];
        }
        return ;
}
```

Có thể cải tiến việc tìm vị trí thích hợp để chèn x_i bằng phép *tìm nhị phân (bài tập).*

d. Độ phức tạp của thuật toán

+ Trường hợp tồi nhất xảy ra khi dãy có thứ tự ngược lại: để chèn x_i cần *i* lần so sánh khóa với $x_{i-1}, ..., x_1, x_0$.

$$SS_{xấu} = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$$

$$HV_{xấu} = \sum_{i=2}^{n} (i+1)/3 = \frac{n(n+3)}{6} - \frac{2}{3}$$

+ Trong trường hợp tốt nhất (khi dãy đã được sắp):

$$HV_{tốt} = \sum_{i=2}^{n} 1/3 = (n-1)/3$$

$$SS_{tốt} = \sum_{i=2}^{n} 1 = n - 1$$

Tóm lại, độ phức tạp thuật toán:

$$T_{tốt}(n) = O(n).$$
$$T_{xấu}(n) = O(n^2).$$

II.3.3. Phương pháp sắp xếp đổi chỗ đơn giản

(phương pháp nổi bọt hay Bubble Sort)

a. Ý tưởng phương pháp:

Duyệt dãy $x_1, x_2, ..., x_n$. Nếu $x_i > x_{i+1}$ thì *hoán vị hai phần tử kề nhau* x_i *và* x_{i+1}. Lặp lại quá trình duyệt (các phần tử "nặng" - hay lớn hơn - sẽ "chìm xuống dưới" hay chuyển dần về cuối dãy) cho đến khi không còn xảy ra việc hoán vị hai phần tử nào nữa.

Ví dụ: Sắp xếp tăng dãy :

44, 55, 12, 42, 94, 18, 06, 67

Viết lại dãy dưới dạng cột, ta có bảng chứa các kết quả sau mỗi bước lặp:

Bước lặp	0	1	2	3	4	5	6
	44	44	12	12	12	12	**06**
	55	12	42	42	18	06	**12**
	12	42	44	18	06	**18**	**18**
	42	55	18	06	**42**	**42**	**42**
	94	18	06	**44**	**44**	**44**	**44**
	18	06	**55**	**55**	**55**	**55**	**55**
	06	67	**67**	**67**	**67**	**67**	**67**
	67	**94**	**94**	**94**	**94**	**94**	**94**

b. Nội dung thuật toán

Để giảm số lần so sánh thừa trong những trường hợp dãy đã gần được sắp trong phương pháp nổi bọt nguyên thủy, ta lưu lại:

- VịTríCuối: là vị trí của phần tử cuối cùng xảy ra hoán vị ở lần duyệt hiện thời
- SốCặp = VịTríCuối -1 là số cặp phần tử cần được so sánh ở lần duyệt sắp tới.

BubbleSort(x, n)

- Bước 1: SốCặp = n -1;
- Bước 2: Trong khi (SốCặp ≥ 1) thực hiện:

```
{ VịTríCuối = 1;
  i = 1;
  Trong khi (i <  SốCặp) thực hiện:
  {    if (xi > xi+1)
           { Hoán vị xi  và  xi+1;
             VịTríCuối = i;
           }
       i = i +1;
  }
  SốCặp = VịTríCuối -1;
}
```

c. Cài đặt thuật toán

```
void BubbleSort(mang x, int n)
{ int  ChỉSốCuối, SốCặp = n -1;
    while (SốCặp > 0)
    { ChỉSốCuối = 0;
      for (int  i = 0;  i<  SốCặp; i++)
          if (x[i] > x[i+1])
              { HoánVị(x[i], x[i+1]);
                ChỉSốCuối = i;
              }
      SốCặp = ChỉSốCuối;
    }
```

```
    return ;
}
```

d. Độ phức tạp của thuật toán nổi bọt

+ Trong trường hợp tồi nhất (dãy có thứ tự ngược lại), ta tính được:

$$HV_{xấu} = SS_{xấu} = \sum_{i=1}^{n-1} (n\text{-}i) = \frac{n(n-1)}{2}$$

+ Trong trường hợp tốt nhất (dãy đã được sắp):

$$HV_{tốt} = \sum_{i=1}^{n-1} 0 = 0$$

$$SS_{tốt} = n - 1$$

Tóm lại, độ phức tạp thuật toán:

$$T_{tốt}(n) = O(n).$$
$$T_{xấu}(n) = O(n^2).$$

II.3.4. Phương pháp sắp xếp đổi chỗ cải tiến *(ShakerSort)*

a. Ý tưởng phương pháp:

Phương pháp sắp xếp *nổi bọt có nhược điểm là:* các *phần tử có trị lớn* được *tìm và đặt đúng vị trí nhanh hơn* các phần tử *có trị bé*. Phương pháp *ShakerSort* khắc phục nhược điểm trên bằng cách *duyệt 2 lượt từ hai phía để đẩy các phần tử nhỏ (lớn) về đầu (cuối) dãy*; với *mỗi lượt*, *lưu lại vị trí hoán vị cuối cùng* xảy ra, nhằm ghi lại các đoạn con cần sắp xếp và tránh các phép so sánh thừa ngoài đoạn con đó.

Ví dụ: Sắp xếp tăng dãy :

44, 55, 12, 42, 94, 18, 06, 67

Viết lại dãy dưới dạng cột, ta có bảng chứa các kết quả sau mỗi bước lặp:

(L,R) = (1,8) (2,7) (3,4) (4,4)

Bước 0	1	2	3
44	**06**	**06**	**06**
55	44	**12**	**12**
12	12	**18**	**18**
42	42	42	**42**
94	55	44	**44**
18	18	**55**	**55**
06	67	**67**	**67**
67	**94**	**94**	**94**

b. Nội dung thuật toán

ShakerSort(x, n)

- Bước 1: L = 1; R = n;
- Bước 2:
  * Bước 2a: *// Duyệt từ dưới lên để đẩy phần tử nhỏ về đầu dãy: L*
    j = R; ChỉSốLưu = R;
    Trong khi (j > L) thực hiện:
    { if ($x_j < x_{j-1}$)
      { Hoán vị x_j và x_{j-1};
        ChỉSốLưu = j;
      }
      j = j -1;
    }
    L = ChỉSốLưu; // Không xét các phần tử đã sắp ở đầu dãy
  * Bước 2b:*// Duyệt từ trên xuống để đẩy phần tử lớn về cuối dãy: R*
    j = L; ChỉSốLưu = L;
    Trong khi (j < R) thực hiện:
    { if ($x_j > x_{j+1}$)
      { Hoán vị x_j và x_{j+1};
        ChỉSốLưu = j;
      }
      j = j +1;
    }
    R = ChỉSốLưu; // Không xét các phần tử đã sắp ở cuối dãy
- Bước 3: if (L < R) Quay lại bước 2;
  else Dừng.

c. Cài đặt thuật toán

```
void ShakerSort(mang x, int n)
{ int ChỉSốLưu,  j, L = 0, R = n-1;
    do
    {// Duyệt từ dưới lên để đẩy phần tử nhỏ về đầu dãy: L
      ChỉSốLưu = R;
      for (j = R; j > L; j--)
      { if  (x[ j ] < x[ j -1])
          { HoánVị(x[ j ], x[ j -1]);
            ChỉSốLưu = j;
          }
      }
      L = ChỉSốLưu; // không xét các phần tử đã sắp ở đầu dãy
      // Duyệt từ trên xuống để đẩy phần tử lớn về cuối dãy: R
      ChỉSốLưu = L;
      for (j = L; j < R; j++)
```

```
    { if  (x[ j ] > x[ j +1])
          { HoánVị(x[ j ], x[ j +1]);
            ChỉSốLưu = j;
          }
    }
   R = ChỉSốLưu; // không xét các phần tử đã sắp ở cuối dãy
  } while (L < R);
  return ;
}
```

d. Độ phức tạp của thuật toán

+ Trong trường hợp tồi nhất (dãy có thứ tự ngược lại), ta tính được:

$$HV_{xấu} = SS_{xấu} = \sum_{i=1}^{n/2} (n\text{-}i) = \frac{n(3n-2)}{8}$$

+ Trong trường hợp tốt nhất (dãy đã được sắp):

$$HV_{tốt} = \sum_{i=1}^{n-1} 0 = 0$$

$$SS_{tốt} = (n\text{ -}1)$$

Tóm lại, độ phức tạp thuật toán:

$$T_{tốt}(n) = O(n).$$
$$T_{xấu}(n) = O(n^2).$$

Phương pháp *ShakerSort* tuy có *tốt hơn Bubble Sort*, nhưng độ phức tạp được cải tiến *không đáng kể*. *Lý do* là hai phương pháp này *chỉ mới đổi chỗ các cặp phần tử liên tiếp* không đúng thứ tự. Nếu các cặp phần tử không đúng thứ tự *ở xa nhau hơn* được đổi chỗ thì độ phức tạp có thể được cải tiến đáng kể như ta sẽ thấy trong phương pháp *QuickSort* sẽ được trình bày ở phần sau.

II.3.5. Phương pháp sắp xếp chèn cải tiến *(ShellSort)*

a. Ý tưởng phương pháp

Một cải tiến của phương pháp chèn trực tiếp là *ShellSort*. Ý tưởng của phương pháp này là phân chia dãy ban đầu thành những dãy con gồm các phần tử ở cách nhau *h* vị trí. Tiến hành sắp xếp từng dãy con này theo phương pháp chèn trực tiếp. Sau đó giảm khoảng cách *h* và tiếp tục quá trình trên cho đến khi *h = 1*.

Ta có thể chọn dãy giảm độ dài $\{h_j\}_{1 \le j \le k}$ thỏa $h_k = 1$ từ hệ thức đệ qui:

$$h_{j-1} = 2* h_j + 1,\ \forall j: 2 \le j \le k = \lfloor \log_2 n \rfloor - 1,\ j=2..k \qquad (1)$$

hoặc:

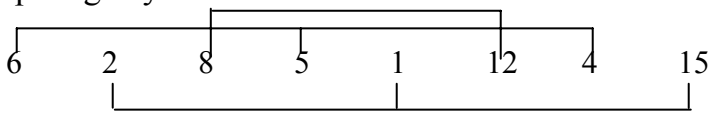
$$h_{j-1} = 3* h_j + 1,\ \forall j: 2 \le j \le k = \lfloor \log_3 n \rfloor - 1,\ j=2..k \qquad (2)$$

b. Nội dung thuật toán

ShellSort(x, n)

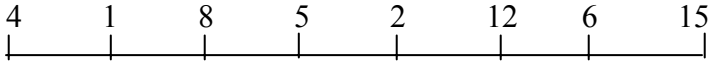
- Bước 1: Chọn k và dãy $h_1, h_2, \ldots, h_k = 1$; j = 1;
- Bước 2: Phân dãy ban đầu thành các dãy con cách nhau h_j khoảng cách. Sắp mỗi dãy con bằng phương pháp chèn trực tiếp.
- Bước 3: j = j +1;
  if (j ≤ k) Quay lại bước 2;
  else Dừng;

* *Ví dụ*: Sắp tăng dãy:



Xét dãy bước: h[1]=3, h[2]= 1 (k=2).

Với h[1] = 3, sắp các dãy con có độ dài 3 bằng phương pháp chèn trực tiếp, ta được:



Với h[2] = 1, sắp các dãy con có độ dài 1 bằng phương pháp chèn trực tiếp như thông thường, ta được:

1 2 4 5 6 8 12 15

c. Cài đặt thuật toán

```
void ShellSort(mang x, int n)
{ int   i, j, k, h[MAX_BUOC_CHIA], len;
  ElemenetType  tam;
      TaoDayBuocChia(n,k,h);   // Xác định  k và dãy h1, h2, ..., hk = 1;
      for (int step = 0; step < k; step++)
      { len = h[step];
        for (i = len; i < n;  i++)
            { tam = x[i];
              j = i - len;  // x[ j ] là phần tử đứng kề trước x[i] trong cùng dãy con
                  // sắp xếp dãy con chứa trị x[i] = tam bằng phương pháp chèn trực tiếp
              while (j >= 0  &&  tam < x[ j ])
              { x[ j + len] = x[j];
                j = j - len;
              }
              x[ j + len] = tam;
            }
      }
      return;
}
```

d. Độ phức tạp của thuật toán

Người ta chứng minh được rằng, nếu chọn dãy bước chia$\{h_j\}$ theo (1) thì thuật toán *ShellSort* có độ phức tạp cỡ: $n^{1,2} << n^2$.

II.3.6. Phương pháp sắp xếp phân hoạch *(QuickSort)*

Phương pháp Quick Sort (hay sắp xếp kiểu phân đoạn) là một *cải tiến* của phương pháp *sắp xếp kiểu đổi chỗ*, do C.A.R. Hoare đề nghị, dựa vào cách *hoán vị các cặp phần tử không đúng thứ tự* có thể ở những vị trí *xa nhau*.

a. Ý tưởng phương pháp:

Chọn một phần tử bất kỳ (ta thường chọn phần tử giữa) g của dãy *làm mốc*. Sau đó thực hiện *phân hoạch dãy thành 2 dãy con: dãy con trái* gồm những phần tử có giá trị *không lớn hơn g* và *dãy con phải* gồm những phần tử có giá trị *không nhỏ hơn g* (bằng cách duyệt dãy từ bên trái cho đến khi có một phần tử $x_i \geq g$, sau đó duyệt dãy từ bên phải cho đến khi có một phần tử $x_j \leq g$. Đổi chỗ x_i và x_j. Tiếp tục quá trình duyệt và đổi chỗ cho tới khi hai phía vượt qua nhau: i > j).

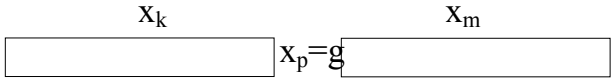
Sau khi phân hoạch, ta tách dãy thành 3 phần:

$x_k \leq g$ với mọi k = 1, ..., j (Dãy con trái hay dãy con thấp);

$x_m \geq g$ với mọi m = i, ..., n (Dãy con phải hay dãy con cao);

$x_p = g$ với mọi p = j+1, ..., i-1, nếu i-1 ≥ j+1.

Vì thế phương pháp này còn gọi là phương pháp sắp xếp bằng phân hoạch. Khi đó, nếu i-1 ≥ j+1 thì các phần tử $x_{j+1}, ..., x_{i-1}$ được định vị đúng:

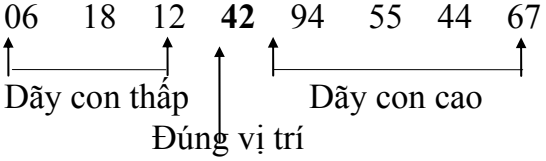


Với *từng dãy con trái và phải (có độ dài lớn hơn 1) ta lại phân hoạch (đệ qui) chúng* tương tự như trên.

Ví dụ: Xét dãy

44 55 12 **42** 94 18 06 67

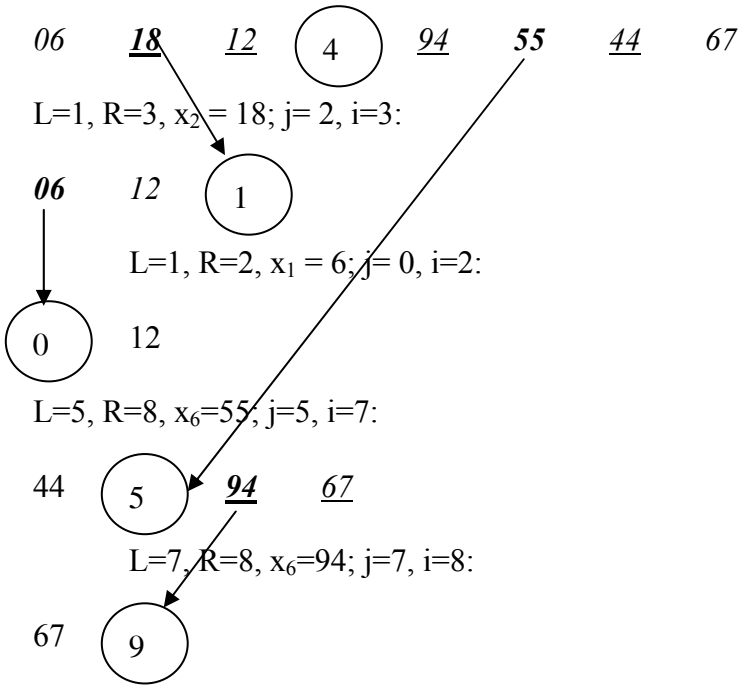
Sau 2 lần đổi chỗ, phân hoạch dãy trên thành



Kết quả phân hoạch qua từng bước đệ qui:

L=1, R=8, x_4=42; j=3, i=5:

44 55 12 ***42*** *94 18 06 67*



Cuối cùng, kết hợp các kết quả đệ qui, ta có dãy được sắp:

06 12 18 42 44 55 67 94

b. Nội dung thuật toán sắp xếp nhanh dãy: $x_L, x_{L+1}, ..., x_R$
SắpXếpNhanh(x, L, R)

- Bước 1: Phân hoạch dãy $x_L, ..., x_R$ thành các dãy con:
  - dãy con thấp: $x_L, ..., x_j \le g$
  - dãy con giữa: $x_{j+1} = ... = x_{i-1} = g$, nếu $i-1 \ge j+1$
  - dãy con thấp: $x_i, ..., x_R \ge g$
- Bước 2: if (L < j) phân hoạch dãy $x_L, ..., x_j$
  if (i < R) phân hoạch dãy $x_i, ..., x_R$

Nội dung thuật toán phân hoạch dãy: $x_L, x_{L+1}, ..., x_R$ thành các dãy con
PhânHoạch(x, L, R)

- Bước 1: Chọn tùy ý một phần tử $g = x_k$;($L \le k \le R$, thường chọn k = (L+R)/2)); i = L; j = R;
- Bước 2: Tìm và hoán vị các cặp phần tử x_i *và* x_j đặt sai vị trí:
  - Trong khi ($x_i < g$) i = i + 1;
  - Trong khi ($x_j > g$) j = j -1;
  - if ($i \le j$)
    { Hoán vị x_i *và* x_j;
    i = i + 1; j = j -1;
    }

- Bước 3: if (i ≤ j) Quay lên bước 2;
  else Dừng;

c. Cài đặt thuật toán

```
void PhânHoạch(mang x, int L, int R)
// L, r : lần lượt là chỉ số trái và phải của dãy con  của mảng  x  cần phân hoạch
{ int  i = L; j = R;
  ElementType giua =  x[(L+R)/2];          // Chọn phần tử  "giữa" làm mốc
        do
        { while (giua>x[i]) i = i+1;
          while (giua<x[j]) j = j-1;
            if (i <= j)
            {  HoánVị(x[i],x[j]);
               i++ ; j-- ;
            }
        } while (i <= j);
        if (L < j)  PhânHoạch(x, L, j);
        if (R  > i)  PhânHoạch(x, i, R);
       return;
}

void SắpXếpNhanh (mang x, int n)
{      PhânHoạch(x, 0, n-1);
       return;
}
```

d. Độ phức tạp của thuật toán

Người ta chứng minh được rằng:

+ Trong trường hợp *xấu nhất* (khi phân hoạch mọi dãy thành hai dãy con, *luôn có một dãy con có độ dài không,* chẳng hạn*, chọn* $g = x_L$ và *dãy ban đầu được sắp* theo *thứ tự ngược lại*):

$$T_{xấu}(n) = O(n^2)$$

nghĩa là, sắp xếp nhanh (*QuickSort) không hơn gì các phương pháp sắp xếp trực tiếp đơn giản,* nhưng *trường hợp này hiếm khi xảy ra:* để *tránh tình trạng này,* ta *thường chọn* $g = x_{giữa}$.

+ Trong trường hợp *tốt nhất:* sau mỗi phân hoạch, ta *đều chọn đúng mốc là* phần tử *median* cho dãy con (phần tử có trị nằm giữa dãy). Khi đó, ta sẽ cần $log_2(n)$ *lần phân hoạch* thì sắp xếp xong. Độ phức tạp trong *mỗi lần phân hoạch là* $O(n)$. *Vậy:* $T_{tốt}(n) = O(nlog_2n)$

+ Trong trường hợp *trung bình* thì *:*

$$T_{tbình}(n) = O(nlog_2n)$$

QuickSort là phương pháp *sắp xếp trong trên mảng rất hiệu quả được biết cho đến nay.*

II.3.7. Phương pháp sắp xếp trên cây có thứ tự *(HeapSort)*

Với phương pháp sắp xếp Quick Sort, thời gian thực hiện trung bình khá tốt, nhưng trong trường hợp xấu nhất nó vẫn là $O(n^2)$. Phương pháp *HeapSort* mà ta sẽ xét sau đây có *độ phức tạp trong trường hợp xấu nhất là* $O(n\log_2 n)$.

Nhược điểm của phương pháp *chọn trực tiếp là ở lần chọn hiện thời không tận dụng được kết quả so sánh và hoán vị của các lần chọn trước đó.* Phương pháp dựa trên *khối HeapSort* khắc phục được nhược điểm này bằng cách đưa dãy cần sắp vào *cây nhị phân có thứ tự (*hay *Heap*) và chúng được lưu trữ kế tiếp bằng mảng.

a. Định nghĩa và tính chất của khối (Heap)

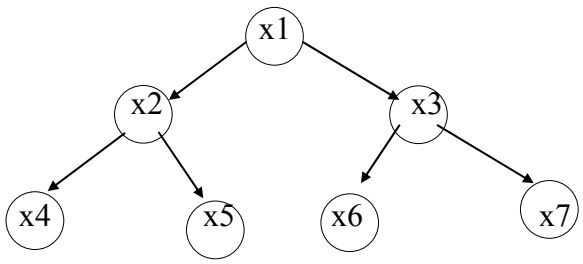
Định nghĩa: Dãy $x_m, ..., x_n$ là một *Heap* nếu :

$$x_k \geq x_{2k},$$
$$x_k \geq x_{2k+1},$$

với mọi k mà : $m \leq k < 2k < 2k+1 \leq n$.

Tính chất:

- Nếu dãy $x_1, ..., x_n$ có thứ tự thì nó là một *Heap*. Chú ý điều ngược lại chưa chắc đúng, nghĩa là: nếu dãy $x_1, ..., x_n$ là một *Heap* thì chưa chắc dãy đã có thứ tự.
- Nếu dãy $x_1, ..., x_n$ là một *Heap* thì x_1 *là phần tử lớn nhất* trong dãy và nếu bỏ đi một số phần tử liên tiếp ở hai đầu của dãy thì nó vẫn là một *Heap*.
- Với dãy bất kỳ $x_1, ..., x_n$ thì dãy $x_{[n/2]+1}, ..., x_n$ (nửa đuôi dãy) là một Heap.
- Nếu dãy $x_1, ..., x_n$ là một *Heap* thì ta có thể biểu diễn "liên tiếp" những phần tử của dãy này lên một cây nhị phân có tính chất: con trái (nếu có) của x_i là $x_{2i} \leq x_i$ và con phải (nếu có) của x_i là $x_{2i+1} \leq x_i$.



…

b. Ý tưởng phương pháp:

Nếu biểu diễn một Heap $x_1, ..., x_n$ lên cây nhị phân có thứ tự, ta sẽ thu được dãy có thứ tự bằng cách :

- Hoán vị nút *gốc* x_1 (*lớn nhất*) với *nút cuối* x_n
- Khi đó $x_2, ..., x_{n-1}$ *vẫn là một heap. Bổ sung* x_1 *vào heap cũ* $x_2, ..., x_{n-1}$ *để được heap mới dài hơn* $x_1, ..., x_{n-1}$.

Lặp lại quá trình trên *cho đến khi cây chỉ còn một nút.*

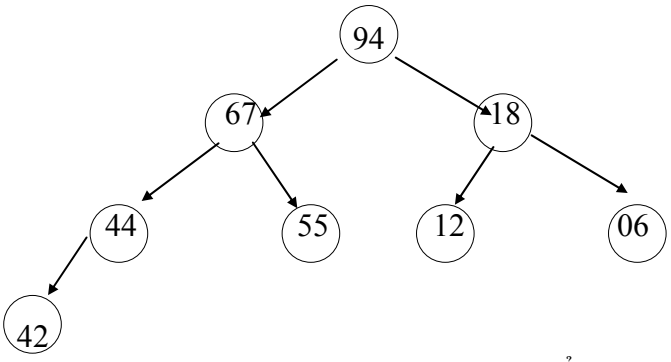
Ví dụ: Sắp xếp dãy số

44 55 12 42 94 18 06 67

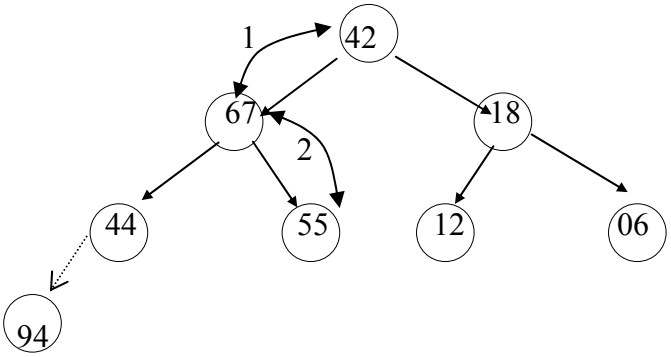
Giả sử tồn tại thủ tục để tạo một Heap đầy đủ ban đầu từ dãy trên :

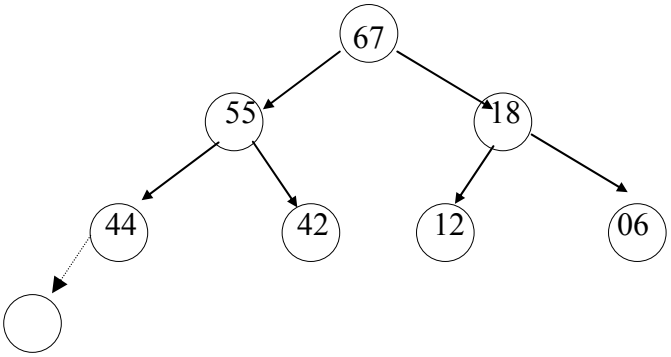
94 67 18 44 55 12 06 42

Cây nhị phân biểu diễn Heap ban đầu



Hoán vị nút 94 với nút 42 và bổ sung 42 vào heap cũ: 67, 18, 44, 55, 12, 06 để được heap mới dài hơn: 67, 55, 18, 44, 42, 12, 06. Để ý rằng, ta *chỉ xáo trộn không quá một nhánh* (nhánh trái có gốc là 67) với gốc (42) của cây cũ.





94

Tiếp tục quá trình trên cho đến khi dãy chỉ còn một phần tử thì ta sẽ được dãy tăng:

06 12 18 42 44 55 67 94

c. Nội dung thuật toán HeapSort

- Giai đoạn 1: Từ Heap ban đầu: $x_{[n/2]+1}, ..., x_n$, *tạo Heap đầy đủ* ban đầu
- Giai đoạn 2: *Sắp xếp dãy dựa trên Heap:*
  - Bước 1: r = n;
  - Bước 2: Đưa phần tử lớn nhất về cuối dãy đang xét: Hoán vị x_1 *và* x_r
  - Bước 3: . Loại phần tử lớn nhất ra khỏi Heap: r = r –1;
    . *Bổ sung* x_1 vào heap cũ: $x_2, ..., x_r$ để được heap mới dài hơn: $x_1, ..., x_r$. *// dùng thủ tục Shift(x, 1, r)*
  - Bước 4: if (r > 1) Quay lên bước 2
    else Dừng //Heap chỉ còn một phần tử

* <u>*Nội dung thuật toán Shift*</u>*:* Bổ sung x_L vào heap cũ: $x_{L+1}, ..., x_r$ để được heap mới dài hơn: $x_L, ..., x_r$.

Shift (x, L, R)

- Bước 1: ChỉSốCha = L; ChỉSốCon = 2* ChỉSốCha; Cha = $x_{ChỉSốCha}$;
  LàHeap = False;
- Bước 2: Trong khi (Chưa LàHeap and ChỉSốCon ≤ R) thực hiện:

```
{ if (ChỉSốCon < R) // nếu Cha có con phải, tìm con lớn nhất
      if (x_ChỉSốCon < x_ChỉSốCon+1)  ChỉSốCon = ChỉSốCon +1;
  if (x_ChỉSốCon ≤ Cha) LàHeap = True;
  else { x_ChỉSốCha  =  x_ChỉSốCon; // đưa nút con lớn hơn lên vị trí nút cha
         ChỉSốCha =  ChỉSốCon;
         ChỉSốCon = 2* ChỉSốCha;
       }
}
```

- Bước 3: $x_{ChỉSốCha}$ = Cha;

c. Cài đặt thuật toán

* <u>*Thủ tục Shift*</u>:

// Thêm x_L vào Heap $x_{L+1}, ..., x_r$ để tạo Heap mới dài hơn một phần tử $x_L, ..., x_r$,

void Shift(mang x, int L, int R)

```
{ int ChỉSốCha = L, ChỉSốCon = 2* ChỉSốCha,  LàHeap = 0;
  ElementType Cha = x[ChỉSốCha];
```

```
        while (!LàHeap  && (ChỉSốCon ≤ R))
         { if (ChỉSốCon < R)   // Chọn nút có khóa lớn nhất trong 2 nút con của nút Cha
                if (x[ChỉSốCon] < x[ChỉSốCon+1])  ChỉSốCon++;
           if (Cha >= x[ChỉSốCon]) LàHeap = 1;
           else { x[ChỉSốCha] = x[ChỉSốCon]; // Chuyển nút con lớn hơn lên nút cha
                  ChỉSốCha = ChỉSốCon;
                  ChỉSốCon = 2* ChỉSốCha;
                }
        }
        x[ChỉSốCha] =  Cha;
        return ;
}
```

Chú ý rằng, với dãy ban đầu bất kỳ $x_{1,} ..., x_{n,}$ thì $x_{[n/2]+1,} ..., x_n$ *là Heap ban đầu (không đầy đủ).* Sau đó áp dụng liên tiếp *thuật toán Shift bổ sung phần tử kề bên trái vào các Heap đã có, ta được các Heap mới nhiều hơn một phần tử ...* Cuối cùng, ta đực *Heap đầy đủ ban đầu:* $x_{1,} ..., x_n$.

* *Tạo Heap đầy đủ ban đầu từ Heap ban đầu của dãy* $x_{1,} ..., x_n$

```
void TạoHeapĐầyĐủ(mang x, int n)
{ int L = n/2,  R = n-1;
        while (L >= 0)  Shift(x, L--, R);
        return ;
}
```

* *Ví dụ*: Từ dãy 44 55 12 42 **94** 18 06 67

Heap ban đầu

L=3	44	55	12	**67**	94	18	06	42
L=2	44	55	**18**	67	94	12	06	42
L=2	44	**94**	18	67	55	12	06	42
L=1	**94**	**67**	18	44	55	12	06	42

Heap đầy đủ đã tạo xong

* *Thủ tục HeapSort*

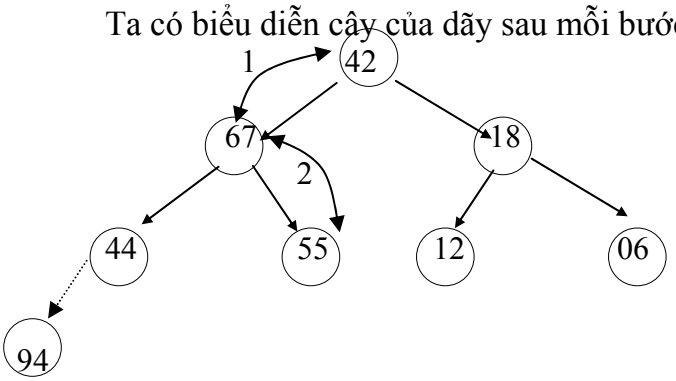
```
void HeapSort(mang x, int n)
{       TạoHeapĐầyĐủ(x, n);
  int L = 0,  R = n -1;
        while (R > 0)
         { HoánVị(x[0], x[R]);
           Shift(x, L, --R);
```

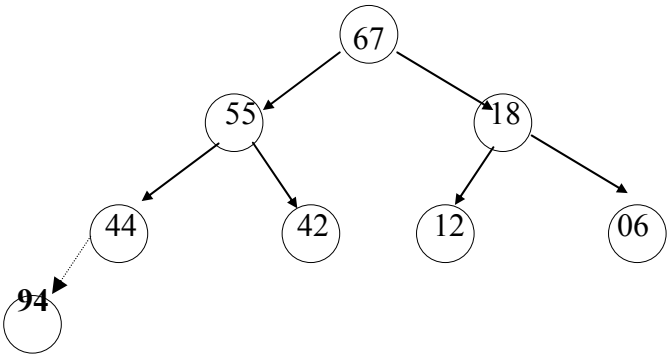
```
        }
    return ;
}
```

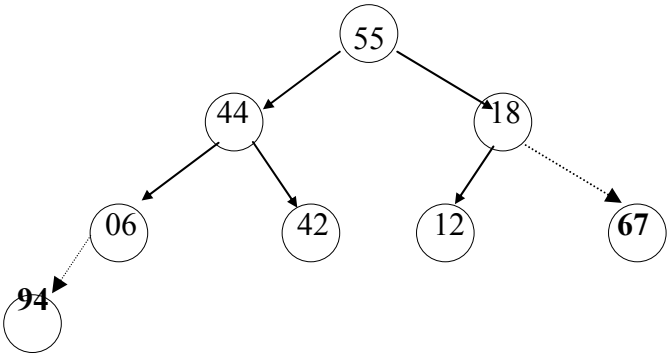
Ví dụ: Với Heap ban đầu:

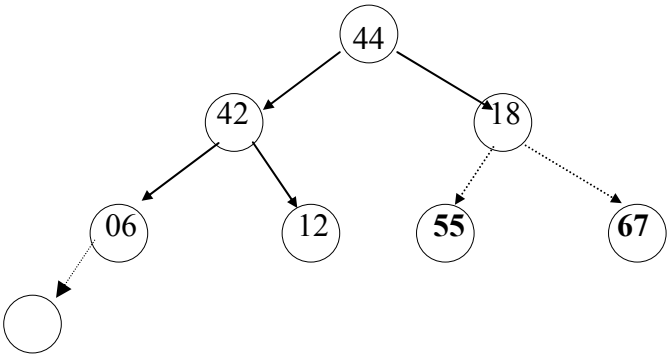
94 67 18 44 55 12 06 42

Ta có biểu diễn cây của dãy sau mỗi bước lặp:

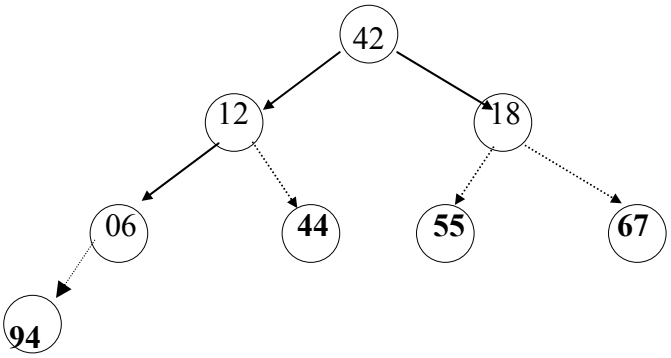


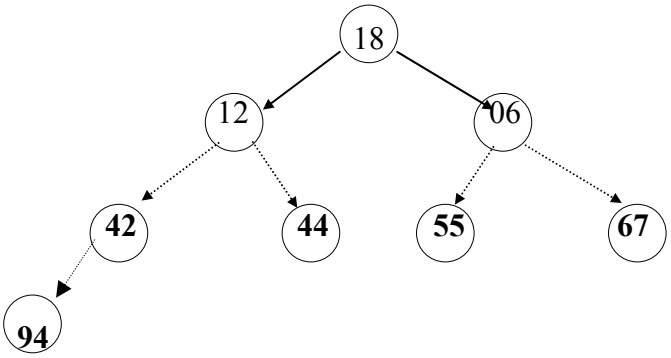


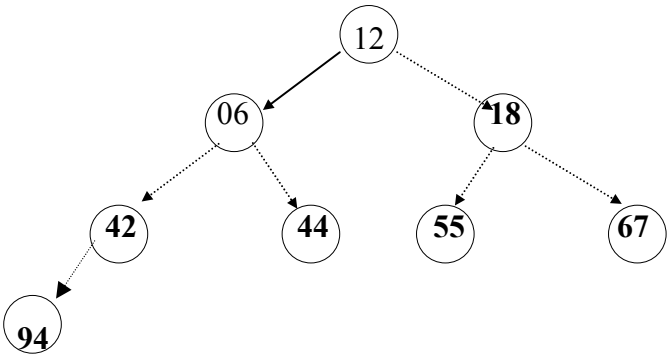


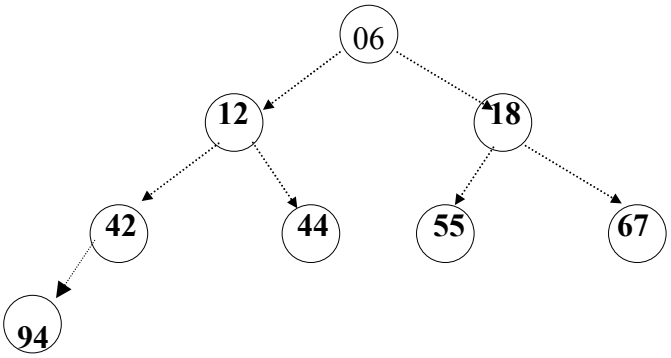


94


42
12
18
06
44
55
67
94


18
12
06
42
44
55
67
94


12
06
18
42
44
55
67
94


06
12
18
42
44
55
67
94

Duyệt các cây theo chiều rộng, ta có kết quả dưới dạng dãy sau mỗi bước lặp:

67	**55**	18	44	**42**	12	06	94
55	**44**	18	**06**	42	12	67	94
44	**42**	18	06	**12**	55	67	94
42	**12**	18	06	44	55	67	94
18	12	**06**	42	44	55	67	94
12	**06**	18	42	44	55	67	94
06	12	18	42	44	55	67	94

d. Độ phức tạp của thuật toán

Người ta chứng minh được rằng trong trường hợp tồi nhất, độ phức tạp của thuật toán Heap Sort là:

$T_{xấu}(n) = O(n\log_2 n)$.

Trong thuật toán đệ quy *QuickSort* cần không gian nhớ cho ngăn xếp (để lưu thông tin về các phân đoạn sẽ được xử lý tiếp theo và do đó sẽ phụ thuộc vào kích cỡ dữ liệu đầu vào). Đối với thuật toán *HeapSort (dưới dạng lặp)*, ta cần *không gian nhớ phụ là hằng (nhỏ) không phụ thuộc vào kích cỡ dữ liệu đầu vào.*

II.3.8. Phương pháp sắp xếp trộn *(Merge Sort)*

a. Ý tưởng phương pháp:

Dựa trên ý tưởng "*chia để trị*", phương pháp sắp xếp trộn được xây dựng dựa vào nhận xét: với mỗi dãy con, ta đều có thể ***tách*** chúng thành *tập các dãy con được sắp*. Nếu ta ***trộn*** *các dãy con (được sắp)* này thì sẽ được các *dãy con (được sắp) dài hơn*, với số lượng dãy con mới ít hơn khoảng một nửa. *Lặp lại* quá trình trên *cho đến khi* tập ban đầu *chỉ còn duy nhất một dãy con*, nghĩa là các phần tử của chúng được sắp xếp.

Trong phương pháp *trộn trực tiếp*, ta xét *các dãy con có chiều dài cố định* 2^{k-1} trong lần tách thứ *k*. Khi đó, ta sẽ *không tận dụng được trật tự tự nhiên* của các dãy con *ban đầu* hay *sau mỗi lần trộn*. Để khắc phục nhược điểm này, ta cần đến khái niệm *đường chạy tự nhiên.* Thay vì trộn các đường chạy có chiều dài cố định ta sẽ *trộn các đường chạy tự nhiên* thành các đường chạy dài hơn.

* *Định nghĩa 1: (đường chạy tự nhiên - với chiều dài không cố định)*

Một *đường chạy (tự nhiên) r* (theo trường khóa *key*) trong dãy *x* là một dãy con được sắp (tăng) lớn nhất gồm các đối tượng $r = \{d_m, d_{m+1}, \ldots, d_n\}$ thỏa các tính chất sau:

$$\begin{cases} d_i.key \le d_{i+1}.key , & \forall\, i \in [m,n) \\ d_{m-1}.key > d_m.key \\ d_n.key > d_{n+1}.key \end{cases}$$

* *Định nghĩa 2: (thao tác trộn)*

Trộn 2 đường chạy r_1, r_2 có chiều dài lần lượt là d_1 và d_2 là tạo ra đường chạy mới r (gồm tất cả các đối tượng từ r_1 và r_2) có chiều dài $d_1 + d_2$.

* *Ví dụ*

Sắp xếp tăng dần bằng phương pháp trộn tự nhiên dãy sau:

x: 75 55 15 20 85 30 35 10 60 40 50 25 45 80 70 65

Các bước tách và trộn trong mỗi bước lặp:

* Tách (lần 1, đưa những đường chạy tự nhiên trong dãy x lần lươt vào các dãy phụ y, z):

y: 75 15 20 85 10 60 25 45 80 65

z: 55 30 35 40 50 70

- Trộn (trộn những đường chạy tự nhiên tương ứng trong các dãy phụ y, z thành các đường chạy mới dài hơn vào dãy x):

x : 55 75 15 20 30 35 40 50 70 85 10 60 25 45 80 65

* Tách (lần 2):

y: 55 75 10 60 65

z: 15 20 30 35 40 50 70 85 25 45 80

- Trộn:

x: 15 20 30 35 40 50 55 70 75 85 10 25 45 60 65 80

* Tách (lần 3):

y: 15 20 30 35 40 50 55 70 75 85

z: 10 25 45 60 65 80

- Trộn:

x: 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85

b. Nội dung thuật toán

TrộnTựNhiên(x, n)

Lặp lại các bước sau:

1. *Gọi thuật toán "Tách" để chia dãy x thành các dãy con và đưa chúng lần lượt vào dãy y và z ;*
2. *Gọi thuật toán "Trộn" để trộn các dãy con trong dãy y và z vào lại x và đếm SốĐườngChạy mỗi khi trộn một cặp đường chạy;*

cho đến khi SốĐườngChạy = 1.

c. Cài đặt thuật toán

Để tiết kiệm bộ nhớ, ta có thể cải tiến thuật toán trên bằng cách chỉ dùng một dãy phụ *y* (có cỡ *n*). (Mỗi khi tách được hai dãy con tự nhiên của dãy *x*, ta đưa chúng vào dãy phụ *y* từ hai phía, sau đó trộn ngay chúng trở lại vào *x).*

void TronTuNhien(mang x, int n)

{ int SoDChay, BDau1, Cuoi1, BDau2, Cuoi2, HếtDãy; // kết thúc dãy *x*

```
  mang y;          // mảng phụ
      do
      { SoDChay = 0; BDau1 = 0; HếtDãy = 0;
        // Tach va tron x thanh cac duong chay tu nhien dai nhat
        while (!HếtDãy)
        { Tim1DChay(x,n -1,BDau1,Cuoi1,HếtDãy); SoDChay++;
          if (!HếtDãy)
                { BDau2=Cuoi1+1;
                  Tim1DChay(x,n -1,BDau2,Cuoi2,HếtDãy);
                  // Trộn 2 dãy con tăng thành dãy con tăng (chỉ dùng một mảng phụ y)
                  Tron(x,y,BDau1,Cuoi1,BDau2,Cuoi2);
                  BDau1 = Cuoi2+1;
                }
        }
      } while (SoDChay>1);
      return;
}
```

// Tìm 1 đường chạy trên *x*, bắt đầu từ chỉ số BDau <= KThuc, trả về chỉ số *Cuối* đường chạy (tăng):
// Neu Cuối < KThuc: HếtDãy = 0; ngược lại, HếtDãy = 1.

int Tim1DChay(mang x, int KThuc, int BDau, int &Cuoi, int &HếtDãy)

```
{ int Truoc = BDau;
        Cuoi = Truoc+1;
        while (Cuoi<=KThuc  &&  x[Truoc] <= x[Cuoi])
           { Truoc = Cuoi;
             Cuoi++;
           }
        if (Cuoi > KThuc)
                { Cuoi = KThuc;
                  HếtDãy = 1; return 1;
                }
        else // x[Truoc] > x[Cuoi]
                { Cuoi--;
                  HếtDãy = 0; return 0;
                }
}
```

//BDau1 <= Cuoi1 < BDau2 = Cuoi1+1 <= Cuoi2

void Tron (mang x, mang y, int BDau1, int Cuoi1, int BDau2, int Cuoi2)

```
{ int k, i, j;
        for (i = Cuoi1; i >= BDau1; i--) y[ i ] = x[ i];
        for (j = BDau2; j <= Cuoi2; j++) y[Cuoi2+BDau2-j] = x[ j ];
        i = BDau1; j = Cuoi2;
        for (k = BDau1; k <= Cuoi2; k++)
            if (y[ i ] < y[ j ])
                {x[k] = y[ i ];  i++;
                }
            else {x[k] = y[ j ]; j--;
```

```
        }
    return;
}
```

Đó là cách tiếp cận *từ dưới lên (Down-Top*) của thuật toán trộn dưới dạng *lặp*. Ta cũng có thể tiếp cận thuật toán trộn theo hướng *từ trên xuống (Top-Down)* dưới dạng *đệ qui (*cho đơn giản và tự nhiên: *bài tập*).

d. Độ phức tạp của thuật toán

- Trong trường hợp *tồi nhất* (khi các mục có thứ tự ngược lại), phương pháp này giống như phương pháp "trộn trực tiếp" (ứng với các đường chạy có độ dài: 1, 2, 4, 8, 16,...). Để sắp xếp một dãy gồm *n* đối tượng, cần đòi hỏi log_2n thao tác "Tách" và mỗi đối tượng trong *n* mục phải được xử lý trong mỗi thao tác. Do đó, độ phức tạp trong trường hợp tồi nhất là:

$$T_{xấu}(n) = O(nlog_2n).$$

- Phương pháp *trộn tự nhiên hiệu qủa về mặt thời gian* nhưng *tốn bộ nhớ phụ cho các dãy phụ.* Dựa trên ý tưởng của phương pháp *trộn tự nhiên, nếu dãy được cài đặt bằng danh sách liên kết (*sẽ trình bày trong chương sau) thì nhược điểm trên sẽ được khắc phục.

- Có thể *cải biên* phương pháp *này để sắp xếp trên bộ nhớ ngoài* (xem giáo trình "Cấu trúc dữ liệu và thuật toán 2").

II.3.9. Phương pháp sắp xếp dựa trên cơ số *(Radix Sort)*

a. Ý tưởng phương pháp

Radix Sort là một phương pháp sắp xếp *không dựa vào việc so sánh trị* của các phần tử như các phương pháp đã trình bày trên đây, mà *dựa vào việc phân loại và trình tự phân loại* sẽ *tạo ra thứ tự* cho các phần tử, tương tự như việc phân loại trước khi phát thư của bưu điện (theo *cây phân cấp* địa phương).

Giả sử các phần tử cần sắp $x_1, ..., x_n$, là *các số nguyên có tối đa m* chữ số. Ta phân loại các phần tử lần lượt theo các chữ số hàng đơn vị, hàng chục, …

b. Nội dung thuật toán

RadixSort(x, n)

- Bước 1: k = 0; // k = 0: hàng đơn vị, k = 1: hàng chục, …
  // *k* cho biết chữ số thứ *k* được dùng để phân loại
- Bước 2: Khởi tạo 10 lô để chứa các phần tử được phân loại:$B_0, ..., B_9$
- Bước 3: Với mỗi *i=1, ..., n*: đặt x_i vào lô B_t (t là chữ số thứ *k* của x_i)
- Bước 4: k = k +1;
  if (k < m) Quay lại bước 2;
  else Dừng;

* *Chú ý*: Sau lần phân phối thứ *k* các phần tử của dãy *X* vào các lô B_0, ..., B_9, rồi lấy các phần tử từ những lô này theo thứ tự của chỉ số *i* của B_i từ 0 đến 9 trở lại X; nếu *chỉ xét k+1 chữ số*, thì *các phần tử của dãy X sẽ tăng*.

* *Ví dụ*: Sắp tăng dãy:

0701 1725 0999 9170 3252 4518 7009 1424 0428 1239 8425

Phân loại dãy vào các lô B theo hàng *đơn vị:*

ChỉSố	Mảng x	0	1	2	3	4	5	6	7	8	9
1	0701	9170	0701	3252		1424	1725			4518	0999
2	1725						8425			0428	7009
3	0999										1239
4	9170										
5	3252										
6	4518										
7	7009										
8	1424										
9	0428										
10	1239										
11	8425										

Phân loại dãy vào các lô B theo hàng *chục:*

ChỉSố	Mảng x	0	1	2	3	4	5	6	7	8	9
1	9170	0701	4518	1424	1239		3252		9170		0999
2	0701	7009		1725							
3	3252			8425							
4	1424			0428							
5	1725										
6	8425										
7	4518										
8	0428										
9	0999										
10	7009										
11	1239										

Phân loại dãy vào các lô B theo hàng *trăm:*

ChỉSố	Mảng x	0	1	2	3	4	5	6	7	8	9
1	0701	7009	9170	1239		1424	4518		0701		0999
2	7009			3252		8425			1725		
3	4518					0428					
4	1424										
5	1725										
6	8425										
7	0428										
8	1239										

9	3252										
10	9170										
11	0999										

Phân loại dãy vào các lô theo hàng *ngàn:*

ChỉSố	Mảng x	0	1	2	3	4	5	6	7	8	9
1	7009	0428	1239		3252	4518			7009	8425	9170
2	9170	0701	1424								
3	1239	0999	1725								
4	3252										
5	1424										
6	8425										
7	0428										
8	4518										
9	0701										
10	1725										
11	0999										

Đưa lần lượt các phần tử của các lô $B_0, ..., B_9$ vào lại dãy X, ta được dãy tăng: 0428 0701 0999 1239 1424 1725 3252 4518 7009 8425 9170

c. Cài đặt thuật toán (*bài tập)*

Chú ý: Do *tổng* các mục dữ liệu trải trên các lô $B_0, ..., B_9$ *luôn bằng n*, nên *cài đặt mỗi lô bằng mảng là không hiệu* quả. Khi đó, nếu *dùng danh sách liên kết động (*xem chương tiếp*)* được cài đặt bởi con trỏ *sẽ hiệu quả hơn*.

d. Độ phức tạp của thuật toán

- Thuật toán *RadixSort* thực hiện *m* lần các thao tác phân bố dãy *X* vào các lô và ghép các lô trở lại dãy *X*. Trong mỗi thao tác này, mỗi phần tử được xét (tính địa chỉ một lần và hai phép gán) đúng một lần. Vậy độ phức tạp của thuật toán (số phép *hoán vị*, trong *cả 3 trường hợp về tình trạng dữ liệu, đều như nhau*) là *cỡ tuyến tính*:

$$T(n) = \frac{2}{3}mn = O(n)$$

- Trên thực tế, thuật toán *cần thêm thời gian để tính toán địa chỉ* (trích chữ số thứ *k* của phần tử nguyên) khi phân lô. Việc cài đặt thuật toán sẽ *thuận tiện hơn* nếu các *phần tử có dạng chuỗi* (*chi phí để trích* ra phần tử thứ *k ít hơn*)
- Thuật toán này sẽ *hiệu quả*, nếu *khóa không quá dài*

II.3.10. So sánh các phương pháp sắp xếp trong

Các phương pháp *sắp xếp trực tiếp* (chọn trực tiếp, nổi bọt, chèn trực tiếp), sắp xếp *ShakerSort,* nói chung, chúng đều có độ phức tạp cỡ *đa thức cấp 2:*

$$T(n) = O(n^2).$$

Phương pháp sắp xếp *ShellSort* có độ phức tạp tốt hơn:

$$T(n) = O(n^{1,2}).$$

Các phương pháp *QuickSort, HeapSort* và *trộn* (tự nhiên) trong hầu hết trường hợp có độ phức tạp tốt hơn nhiều:

$$T(n) = O(n\log_2 n)$$

Khác với cách tiếp cận của các phương pháp sắp xếp trên là dựa vào phép so sánh khoá, phương pháp sắp xếp theo cơ số *RadixSort không dựa trên phép so sánh* khóa mà *dựa vào việc phân loại các chữ số trong mỗi số của dãy số có tối đa m chữ số*. Khi đó, các *phép toán cơ bản* là *lấy ra chữ số thứ k* ($1 \le k \le m$) của mỗi số và *phép gán* các phần tử số. *RadixSort* có độ phức tạp là:

$$T(n) = O(nm) = O(n)$$

* Các *số liệu thực nghiệm* về *thời gian* (đơn vị là *sao*) chạy các thuật toán đã trình bày trên máy PC- Pentium III, 600MHz, 64 MB-RAM, theo các bộ số liệu (dãy các số nguyên dương) cỡ: *n = 130.000* và xét tình trạng dữ liệu trong 3 trường hợp: dãy *ngẫu nhiên* có phân bố đều, dãy đã *được sắp* theo thứ tự *thuận* và *ngược*.

	Ngẫu nhiên			*Thứ tự thuận*			*Thứ tự ngược*		
P.Pháp \ **n**	130000	Chậm	Nhanh	130000	Chậm	Nhanh	130000	Chậm	Nhanh
Chọn trực tiếp	23 909	x		23 794	X		30 029	x	
Chèn trực tiếp	11 326	x		6		X	32 384	x	
Nổi bọt	***65 144***	***X***		*0*		*X*	***92 741***	***X***	
Shaker Sort	39 689	X		0		X	59 215	X	
Shell Sort	*33*		*X*	*11*		*X*	*11*		*X*
Heap Sort	*16*		*X*	*11*		*X*	*11*		*X*
Quick Sort	**11**		**X**	**5**		**X**	**5**		**X**
Trộn tự nhiên	*27*		*X*	*5*		*X*	*22*		*X*
Radix Sort	*286*		*x*	*264*		*x*	*253*		*x*

- Với bộ dữ liệu khá lớn gồm *n = 5.000.000* số nguyên, ba phương pháp QuickSort, HeapSort và ShellSort tỏ ra xứng đáng là "đại diện" tốt cho 3 nhóm phương pháp sắp xếp chính đã nêu ở trên (nó nhanh hơn hẳn so với các phương pháp khác trong cùng nhóm).

Để ý rằng, *cả 3 phương pháp đại diện này đều dựa trên ý tưởng "chia đôi"* ("chia để trị"). Với 3 phương pháp đại diện này, ta có kết qủa thực nghiệm như sau:

	Ngẫu nhiên			*Thứ tụ thuận*			*Thứ tụ ngược*		
P.Pháp \ **n**	$5*10^6$	Chậm	Nhanh	$5*10^6$	Chậm	Nhanh	$5*10^6$	Chậm	Nhanh
Shell Sort	***1862***	***X***		***643***	***X***		***698***	***X***	
Heap Sort	*1571*		*X*	*516*	*X*		*561*		*X*
Quick Sort	**489**		**X**	***291***		***x***	**297**		**X**
NaturalMergeSort	***1851***	***X***		**22**		**X**	***1049***	***X***	

Trên thực tế, với nhiều cơ sở dữ liệu lớn, số lần phải sắp xếp những bộ dữ liệu ngẫu nhiên thường ít. Ta thường gặp tình huống phải *sắp xếp lại* các bộ dữ liệu "*gần được sắp*" sau một số lần cập nhật trên bộ dữ liệu đã được sắp trước đó. Khi đó, *QuickSort* và *sắp trộn tự nhiên* là hai phương pháp đáng lưu ý. Đặc biệt, thuật toán *sắp trộn tự nhiên* còn được sử dụng *hiệu quả trên bộ nhớ ngoài*.

Chương III

CẤU TRÚC DANH SÁCH LIÊN KẾT

III.1. Giới thiệu kiểu dữ liệu con trỏ

III.1.1. So sánh kiểu dữ liệu tĩnh và kiểu dữ liệu động

Do đặc điểm và hạn chế của các *kiểu* dữ liệu *cơ sở* và *kiểu có cấu trúc đơn giản* đã xét *(*gọi là *kiểu dữ liệu tĩnh)* là tính *cố định* và cứng nhắc do *không thay đổi được kích thước* và *cấu trúc* trong chu trình sống, (mặc dù các thao tác trên chúng có thể nhanh và thuận tiện trong một số tình huống); vì vậy, nó khó mô tả một cách thật tự nhiên và đúng bản chất của thực tế vốn sinh động và phong phú.

Khi xây dựng chương trình, nếu cần biểu diễn các đối tượng có *số lượng ổn định* và *có thể dự đoán trước kích thước* của chúng, ta có thể sử dụng biến không động (biến tĩnh hay nửa tĩnh). Chúng thường được *khai báo tường minh* được *truy xuất trực tiếp bằng* một *định danh rõ ràng (*tương ứng với *địa chỉ vùng nhớ lưu trữ biến* này*)*, *tồn tại trong phạm vi khai báo* và *chỉ mất khi ra khỏi phạm vi* này, được *khai báo trong* vùng *Data segment* (vùng dữ liệu) hoặc trong vùng *Stack segment* (biến cục bộ) và có *kích thước không đổi trong suốt phạm vi sống*.

Kiểu dữ liệu tĩnh (và do đó cả *các thao tác cơ bản* tương ứng) sẽ khó:

- biểu diễn, cài đặt và xác định kích thước của các *kiểu dữ liệu đệ qui;*
- cài đặt một cách hiệu quả và tự nhiên (mặc dù nó có thể đơn giản) các đối tượng dữ liệu có *số lượng* các phần tử *khó dự đoán trước* và *biến động nhiều* trong quá trình sống (có thể do các thao tác *thêm vào* và *loại ra* xảy *ra thường xuyên*). Khi đó, nhiều *thao tác cơ bản* trên chúng sẽ *phức tạp, kém tự nhiên,* làm chương trình trở nên khó đọc, khó bảo trì cũng như việc *sử dụng bộ nhớ kém hiệu quả* (do thiếu hay lãng phí bộ nhớ quá nhiều);
- biểu diễn hiệu quả (do *sử dụng bộ nhớ kém hiệu quả*) các *đối tượng dữ liệu lớn chỉ tồn tại nhất thời hay không thường xuyên* trong quá trình hoạt động của chương trình.

Đối với các kiểu dữ liệu có đặc tính: *số lượng biến động*, *kích thước thay đổi* hay *chỉ tồn tại nhất thời* trong chu trình sống, … trong nhiều trường hợp nếu dùng *kiểu dữ liệu động* để biểu diễn sẽ *đúng bản chất* và *tự nhiên hơn* cũng như *thuận lợi hơn trong các thao tác* tương ứng trên chúng.

Trong chương này, ta sẽ xét một kiểu dữ liệu động đơn giản nhất là danh sách liên kết.

III.1.2. Kiểu dữ liệu con trỏ

a. Định nghĩa

Cho trước một kiểu $T = <V, O>$. Kiểu con trỏ **PT** tương ứng với kiểu *T* là kiểu:

PT = <Vp, Op>

trong đó:

- ***Vp*** chứa các địa chỉ lưu trữ các đối tượng kiểu *T* hoặc là **NULL** (*NULL* là một địa chỉ đặc biệt tượng trưng cho một giá trị không quan tâm, thường được dùng để chỉ *địa chỉ "kết thúc"*);

- ***Op*** chứa các thao tác liên quan đến việc định địa chỉ của một đối tượng có kiểu *T* thông qua *con trỏ tương ứng chứa địa chỉ* của đối tượng đó. Chẳng hạn, thao tác tạo một con trỏ chứa địa chỉ một vùng nhớ để lưu trữ một đối tượng có kiểu *T*.

Nói một cách khác, kiểu con trỏ tương ứng với kiểu *T* là một *kiểu dữ liệu của các đối tượng dùng để chứa địa chỉ* vùng nhớ cho các đối tượng có kiểu *T*.

Đối tượng dữ liệu thuộc kiểu con trỏ tương ứng với kiểu T (hay gọi tắt là đối tượng *con trỏ* kiểu *T)* là đối tượng dữ liệu mà giá trị của nó là địa chỉ vùng nhớ của một đối tượng dữ liệu có kiểu T hoặc là trị đặc biệt *NULL*. Khi nói đến đối tượng con trỏ kiểu *T*, ta để ý đến *hai thuộc tính* sau:

(kiểu dữ liệu T, địa chỉ của một đối tượng dữ liệu có kiểu *T*)

Thông tin về *kiểu dữ liệu T* nhằm giúp *xác định dung lượng vùng nhớ* cần thiết để lưu trị của một biến có kiểu T.

Đối tượng dữ liệu con trỏ nhận trị nguyên không âm có kích thước qui định sẵn tùy thuộc vào môi trường hệ điều hành làm việc và ngôn ngữ lập trình đang sử dụng (chẳng hạn, với ngôn ngữ lập trình C, biến con trỏ có kích thước 2 hoặc 4 bytes cho môi trường 16 bits và có kích thước 4 hoặc 8 bytes cho môi trường 32 bits tùy vào con trỏ near (chỉ lưu địa chỉ offset) hay far (lưu cả địa chỉ offset và segment)).

b. Khai báo *(trong C hay C++)*

Kiểu và biến con trỏ được khai báo theo cú pháp sau:

typedef KiểuCơSởT *KiểuConTrỏ;
KiểuConTrỏ BiếnConTrỏ;

hoặc khai báo trực tiếp biến con trỏ thông qua kiểu cơ sở *T*:

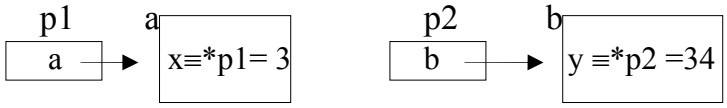
KiểuCơSởT *BiếnConTrỏ, BiếnCơSởT;

KiểuCơSởT có thể là kiểu cơ sở, kiểu dữ liệu có cấu trúc đơn giản, kiểu file hoặc thậm chí là kiểu con trỏ khác. Ngoài ra, ta còn có các *cấu trúc tự trỏ, con trỏ hàm*. Có thể *dùng con trỏ để truyền tham đối cho hàm.*

```
* Ví dụ: typedef  int  *kieu_con_tro_nguyen;                  // cách 1
        kieu_con_tro_nguyen  bien_con_tro_nguyen_2,  p2;
        int *bien_con_tro_nguyen_1, *p1, x, y;        // cách 2: trực tiếp
  p1 = &x;       ( & trong &biến_x  là toán tử  lấy địa chỉ bắt đầu của một biến_x)
  *p1  =  3;
(* trong *p1 là toán tử lấy nội dung trị của biến do p1 trỏ đến, khi đó  x=*p1=3)
  y =  34;
```

p2 = &y; // khi đó *p2 = y = 34

Giả sử a, b lần lượt là địa chỉ bắt đầu của vùng nhớ lưu trị của các biến nguyên x và y tương ứng.



Khi đó, ta nói :

. p1, p2 là hai biến con trỏ kiểu nguyên trỏ đến hai biến kiểu nguyên x và y.

. *p1, *p2 là nội dung của hai biến nguyên x, y mà p1 và p2 trỏ tới.

c. Các thao tác trên kiểu dữ liệu con trỏ

Giả sử ta có khai báo:

*KiểuCơSởT *BiếnConTrỏ_1, *BiếnConTrỏ_2, BiếnCơSởT;*

- Toán tử *gán địa chỉ cho biến con trỏ*:

BiếnConTrỏ = địa_chỉ;

Đặc biệt, địa chỉ này có thể là *NULL. C*ó thể *gán hằng NULL cho bất kỳ biến con trỏ* nào.

BiếnConTrỏ_1 = BiếnConTrỏ_2;

BiếnConTrỏ = &BiếnCơSởT;

trong đó: **&** là toán tử lấy địa chỉ của biến *BiếnCơSởT* có kiểu *KiểuCơSởT,* khi đó ta nói: *BiếnConTrỏ trỏ đến (*hay *chỉ đến) BiếnCơSởT;*

BiếnConTrỏ = địa_chỉ + trị_nguyên;

- Toán tử *truy xuất nội dung* của đối tượng do biến con trỏ *BiếnConTrỏ* trỏ đến:

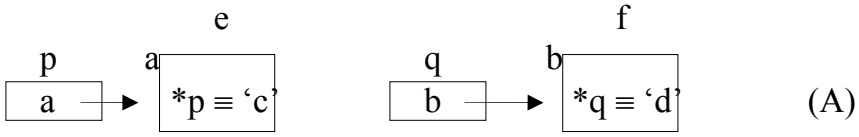
**BiếnConTrỏ*

Khi đó, nếu *BiếnConTrỏ = &BiếnCơSởT* thì **BiếnConTrỏ ≡ BiếnCơSởT.*

* <u>*Ví dụ*</u>: Giả sử cho hai biến con trỏ p, q trỏ đến hai biến kiểu ký tự e, f . Biến e, f có địa chỉ bắt đầu lần lượt là a, b:

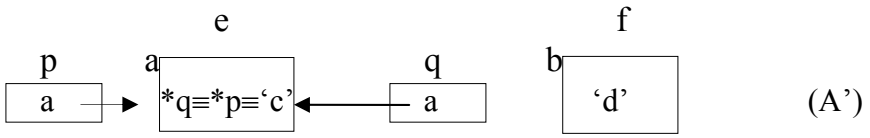
```
char  e, f, *p, *q;
e = 'c'; f = 'd';
p = &e;  q = &f;       // giả sử  p, q có nội dung lần lượt là  a và b
```

Ta có sơ đồ (1) sau đây:

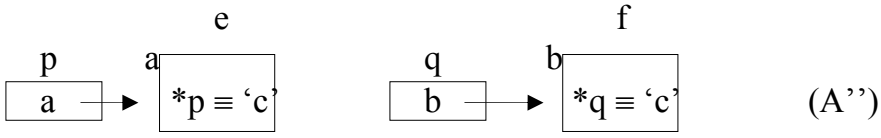


(A)

* Sau lệnh gán hai con trỏ cùng kiểu *q = p* của sơ đồ (A) ta có sơ đồ (A') thay đổi như sau:



* Sau lệnh gán hai biến do hai con trỏ cùng kiểu chỉ đến **q = *p* của sơ đồ (A) ta lại có sơ đồ (A'') thay đổi như sau:



Hãy kiểm tra lại kết quả của các dãy lệnh trên một chương trình trong C++ (*bài tập*).

III.1.3. Biến động

Khi xây dựng các kiểu dữ liệu để biểu diễn các đối tượng trong một bài toán cụ thể, dựa trên các đặc điểm của chúng, nếu ta *không thể dự đoán* hay *xác định trước kích thước* của chúng (do sự *tồn tại, phát sinh* và *mất đi* của chúng *tùy thuộc vào ngữ cảnh* của chương trình hoặc vào người sử dụng chương trình) thì ta có thể sử dụng *biến động* để biểu diễn chúng.

a. Đặc trưng của biến động (hay *biến được cấp phát động*):

- *không* được *khai báo tường minh (không có tên);*
- được *cấp phát bộ nhớ (trong vùng Heap segment)* hoặc *giải tỏa vùng nhớ* đã chiếm dụng (để về sau có thể sử dụng lại vùng nhớ này cho các mục đích khác) *theo yêu cầu* của người sử dụng *khi chương trình đang thi hành* (chứ không phải ở thời điểm biên dịch chương trình). Vì vậy, chúng *không tuân theo qui tắc phạm vi* như biến tĩnh;
- *Số lượng các biến động* có thể *thay đổi* trong quá trình sống (khi chương trình đang thi hành).

b. Truy xuất biến động

Khi *biến động được tạo ra* (cấp phát vùng nhớ để lưu trữ chúng), ta phải *dùng một biến con trỏ* (biến không động và có định danh rõ ràng) *BiếnConTrỏ* có kiểu tương ứng để *lưu giữ địa chỉ* bắt đầu của vùng nhớ này. Sau đó, ta có thể *truy xuất đến biến động thông qua biến con trỏ* đó:

**BiếnConTrỏ*

Nếu dùng biến con trỏ *p* chỉ đến một biến động có kiểu cấu trúc với các thành phần $\{Field_i\}_{1 \le i \le m}$ thì ta có thể truy cập đến thành phần thứ *i*: $Field_i$ của biến động đó thông qua con trỏ p như sau:

p->Field$_i$

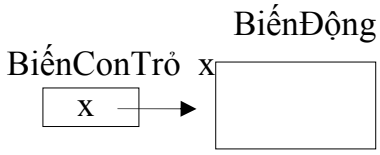
hoặc: (*p).$Field_i$

c. Hai thao tác cơ bản trên biến động: *tạo* và *hủy* một biến động do biến con trỏ trỏ đến.

* *Tạo* một biến động do biến con trỏ trỏ đến: bằng cách *cấp phát vùng nhớ* (địa chỉ bắt đầu và kích thước vùng nhớ tương ứng với kiểu) *cho biến động để lưu trữ đối tượng* và ta *dùng một biến con trỏ để lưu giữ địa chỉ vùng nhớ* đó.

Trong C++, ta dùng hàm *new* để cấp phát vùng nhớ cho một biến động có kiểu cơ sở *T* theo cú pháp sau:

BiếnConTrỏ = ***new*** *KiểuCơSởT;* *// (1)*



Khi đó, ta có thể truy xuất đến (*nội dung*) biến động (không có định danh riêng) thông qua biến con trỏ như sau: **BiếnConTrỏ.*

Hàm new còn có một cách sử dụng khác là:

BiếnConTrỏ = ***new*** *KiểuCơSởT* ***[SốLượng]*** *; // (2)*

để cấp phát vùng nhớ cho *SốLượng* đối tượng có cùng kiểu *KiểuCơSởT* mà *địa chỉ bắt đầu của vùng nhớ này được lưu giữ trong biến con trỏ BiếnConTrỏ.*

Khi đó: địa chỉ bắt đầu vùng nhớ của đối tượng được cấp phát động thứ *i (0 ≤ i ≤ SốLượng -1)* được truy xuất bởi:

BiếnConTrỏ + i

và nội dung của đối tượng được cấp phát động thứ *i (0 ≤ i ≤ SốLượng -1)* được truy xuất bởi:

****(BiếnConTrỏ + i)*** hoặc ***BiếnConTrỏ[i]***

Cú pháp truy xuất trên cũng đúng với "*mảng động"* đã biết:

```
ptử  *BiếnMảngĐộng;
BiếnMảngĐộng = new ptử [MAX];
```

* *Hủy* một biến động đã được cấp phát bởi toán tử *new* do biến con trỏ trỏ đến:

Để giải tỏa vùng nhớ của biến động đã được cấp phát trước đó bằng toán tử *new* do biến con trỏ *BiếnConTrỏ* trỏ đến, ta dùng toán tử *delete* trong C++ như sau:

delete *BiếnConTrỏ*;

hoặc: **delete [ ]*BiếnConTro*;**

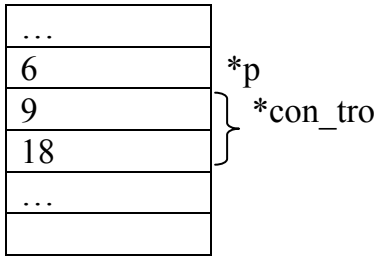
tương ứng với toán tử cấp phát vùng nhớ ***new*** ở dạng (1) hoặc (2) ở trên.

* *Ví dụ*:

```
typedef struct { int   diem;
                 int   tuoi;
               } hs;
```

```
hs *con_tro;
int  *p, *q;

p = new int;
*p = 6;
con_tro =  new hs;
con_tro->diem = 9;  // hoặc:          (*con_tro).diem = 9;
con_tro->tuoi =  18;
```

Minh họa một phần bộ nhớ Heap segment:

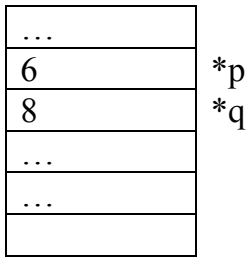


Sau đó thi hành các lệnh:

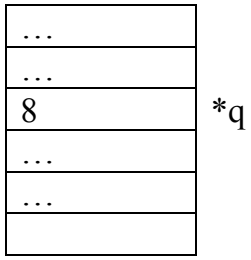
```
delete  con_tro;  // giải toả vùng nhớ do con_tro chiếm giữ
q = new int;
```

Khi đó *q có thể* trỏ đến vùng nhớ do biến *con_tro* trước đây trỏ đến.

```
*q = 8;
```




```
delete p;
```



Dựa trên kiểu dữ liệu động cơ sở là con trỏ, ta có thể xây dựng các kiểu dữ liệu động phong phú khác có nhiều ứng dụng trên thực tế như: danh sách liên kết động, cấu trúc cây, đồ thị, …

III.2. Danh sách liên kết (DSLK)

III.2.1. Định nghĩa danh sách

Cho kiểu dữ liệu *T*. Kiểu dữ liệu danh sách *TL* gồm các phần tử thuộc kiểu *T* được định nghĩa là:

TL = <VL, OL >

với:

- ***VL*** là tập các phần tử có kiểu *T* được móc nối theo kiểu *thứ tự tuyến tính*.
- ***OL*** gồm các toán tử: tạo danh sách, duyệt danh sách, tìm một đối tượng (thỏa một tính chất nào đó) trên danh sách, chèn một đối tượng vào danh sách, hủy một đối tượng khỏi danh sách, sắp xếp danh sách theo một quan hệ thứ tự nào đó, …

III.2.2. Các cách tổ chức danh sách

Có *hai cách* chính để tổ chức danh sách tùy thuộc vào cách tổ chức trình tự tuyến tính các phần tử của danh sách theo kiểu *ngầm* hay *tường minh*.

Ta có thể tổ chức trình tự tuyến tính theo *kiểu ngầm thông qua chỉ số (*như *mảng* hay *file)*. Phần tử x_{i+1} được xem là phần tử kề sau của x_i. Với cách này, các phần tử của danh sách sẽ *được lưu trữ liên tiếp trong một vùng nhớ liên tục*. Việc *truy nhập các phần tử* được thực hiện thông qua *công thức dịch địa chỉ* để xác định *địa chỉ bắt đầu* của phần tử thứ *i (nếu phần tử đầu tiên được đánh số là 0)*:

Địa chỉ bắt đầu danh sách + i(kích thước của T)*

Áp dụng cách tổ chức này, *mảng* có *hạn chế* là *số phần tử tối đa của mảng bị giới hạn cố định (*vùng nhớ được cấp phát liên tục cho mảng được thực hiện khi *biên dịch* đoạn chương trình chứa khai báo biến mảng đó*);* do đó việc *sử dụng bộ nhớ sẽ ít linh động và kém hiệu quả*. Ngoài ra, các thao tác *thêm* và *hủy* sẽ bất tiện và *chiếm nhiều thời gian* để dời chỗ các dãy con của danh sách. Bù lại, việc *truy xuất trực tiếp* các phần tử của mảng trên vùng nhớ liên tục sẽ *nhanh*.

Để *khắc phục các hạn chế* trên, ta có thể tổ chức danh sách tuyến tính theo kiểu móc nối (hay liên kết và gọi là *danh sách liên kết*) ở dạng *tường minh*: *mỗi phần tử ngoài thành phần thông tin về dữ liệu* còn chứa *thêm liên kết (địa chỉ) đến phần tử kế* tiếp trong danh sách. Khi đó, các phần tử của danh sách không nhất thiết phải được lưu trữ kế tiếp trong một vùng nhớ liên tục. Tuy nhiên, do việc truy xuất đến các phần tử của danh sách là *tuần tự*, nên một số thuật toán trên danh sách được cài đặt theo kiểu liên kết sẽ bị chậm hơn.

Sau đây, ta sẽ chủ yếu tập trung khảo sát các kiểu danh sách liên kết động được cài đặt bởi con trỏ: DSLK đơn (có hoặc không có nút câm), DSLK đối xứng, DSLK vòng, DSLK đa liên kết và một số ứng dụng của chúng.

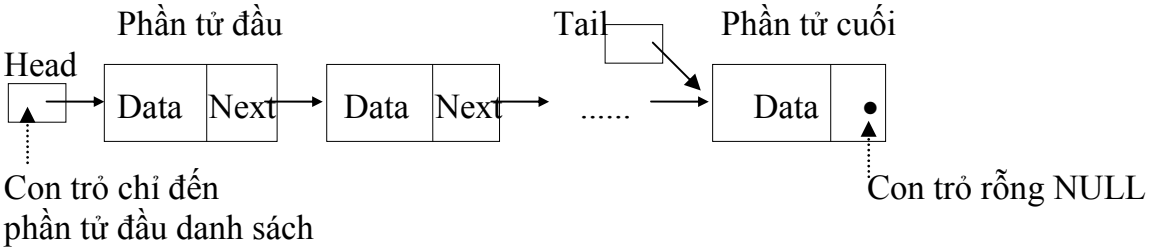
III.3. DSLK đơn

III.3.1. Tổ chức DSLK đơn, các thao tác cơ bản, tìm kiếm và sắp xếp trên DSLK đơn

a. Tổ chức DSLK đơn (không có nút câm)

Mỗi phần tử (còn được gọi là *nút*) của danh sách chứa *hai thành phần* :

- Thành phần *dữ liệu Data*: chứa thông tin *dữ liệu của bản thân phần tử.*
- Thành phần *liên kết Next*: chứa *địa chỉ của nút kế tiếp* trong danh sách hoặc trị *NULL* đối với *nút cuối* danh sách.



Để truy cập đến các phần tử của DSLK, ta *chỉ cần biết* địa chỉ *Head* của nút dữ liệu đầu tiên. Sau đó, khi cần thiết, theo trường *Next* ta có thể biết được địa chỉ (và do đó, nội dung dữ liệu) của nút kế tiếp.

Khi biết nút đầu *Head*, để truy nhập đến nút cuối của danh sách, ta cần *chi phí O(n)* để duyệt qua lần lượt tất cả *n* nút của nó. Mặt khác, để thao tác *tìm kiếm tuần tự* (rất thường gặp khi khai thác thông tin) được hiệu quả, ta thường sử dụng *thêm lính canh ở cuối* danh sách. Vì vậy, để *chi phí việc truy nhập đến nút cuối* là hằng O(1), khi quản lý DSLK, ngoài việc lưu trữ (địa chỉ) *nút đầu Head*, ta còn lưu thêm (địa chỉ) *nút cuối Tail*.

* *Biểu diễn danh sách liên kết (bằng con trỏ)*

- *Trong C hay C++,* mỗi nút của DSLK được cài đặt bởi cấu trúc sau:

```
typedef    ....   ElementType;          // Kiểu dữ liệu cơ sở của mỗi phần tử
typedef struct node {ElementType  Data;
                     struct node *Next;
                    }NodeType;
typedef  NodeType *NodePointer;
typedef  struct { NodePointer Head, Tail;
                } LL;
```

LL List;

- *Trong PASCAL,* mỗi nút của DSLK được cài đặt bởi cấu trúc sau:

```
Type    ElementType = ....;                 // Kiểu dữ liệu cơ sở của mỗi phần tử
        NodePointer = ^NodeType;
NodeType   = record Data: ElementType;
                                    Next: NodePointer;
                    end;
        LL          =  record Head: NodePointer;
                              Tail: NodePointer;
                    end;
var List : LL;
```

Ngoài việc dùng kiểu dữ liệu con trỏ, ta còn có thể biểu diễn một DSLK bằng mảng như sau:

```
#define MAXSIZE  ...                    // Kích thước tối đa của mảng
typedef  .....  ElementType;            // Kiểu dữ liệu của nút
typedef  unsigned int IndexType;        // Miền chỉ số của nút
typedef struct { ElementType  Data;
                 IndexType  Next;
               } NodeType;
typedef  NodeType  Table [MAXSIZE];
typedef  struct { Table  DS;
                  IndexType StartIndex;
                } Table_List;
```

Những thao tác cơ bản trên DS với kiểu cài đặt này là đơn giản (xem như bài tập). Cách cài đặt này gặp hạn chế do kích thước của mảng cố định.

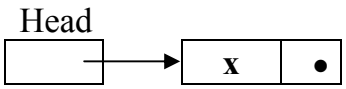
b. Các thao tác cơ bản trên kiểu DSLK đơn

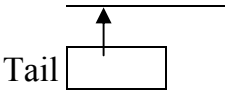
Để tiện theo dõi và thống nhất trong trình bày, ta qui ước các khai báo sau:

```
ElementType  x;          // x là dữ  liệu chứa trong một nút
NodePointer    new_ele; // new_ele là biến con trỏ chỉ đến nút mới được cấp phát
```

Để việc trình bày phần cài đặt các thao tác cơ bản được gọn hơn, ta sẽ sử dụng thủ tục cấp phát động bộ nhớ cho một nút của DSLK sau đây:

Cấp phát vùng nhớ chứa dữ liệu x cho một nút của DSLK





- ***Thuật toán***

  NodePointer ***CreateNodeLL*** *(x)*

  . Cấp phát vùng nhớ cho một nút new_ele;

  . new_ele ->Data = x;

  . new_ele ->Next = NULL;

- ***Cài đặt***

```
NodePointer  CreateNodeLL (ElementType x)
{ NodePointer   new_ele;
        if ((new_ele = new NodeType) ==NULL)
                cout << "\nLỗi cấp phát vùng nhớ cho một nút mới !";
        else    { Gán(new_ele ->Data, x); new_ele ->Next = NULL;
                }
        return new_ele;
}
```

- ***Khởi tạo một DSLK rỗng.***
- ***Thuật toán***

  LL ***CreateEmptyLL*** *()*

  List.Head = List.Tail = NULL;

- ***Cài đặt***

```
LL CreateEmptyLL ()
{ LL List;
        List.Head = List.Tail = NULL;
        return List;
}
```

- ***Kiểm tra một DSLK có rỗng hay không***
- ***Thuật toán***

  Boolean ***EmptyLL****(LL List)*

  if (List.Head == NULL)

  // hay chặt chẽ hơn (List.Head == NULL) && (List.Tail == NULL)

  Trả về trị True; // List rỗng;

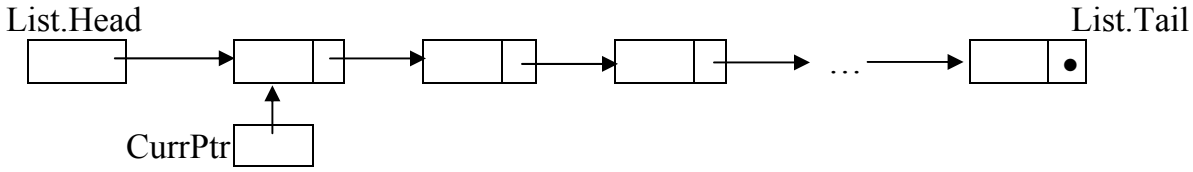
  else Trả về trị False; // List khác rỗng;

- ***Cài đặt***

```
int EmptyLL(LL  List)
```

```
{       return(List.Head == NULL);
  // hay chặt chẽ hơn return ((List.Head == NULL) && (List.Tail == NULL));
}
```

- ***Duyệt qua một DSLK***: Duyệt là đi qua mọi phần tử của DSLK theo một quy luật nào đó (chẳng hạn, từ đầu đến cuối) và mỗi phần tử được xử lý đúng một lần.



- ***Thuật toán***

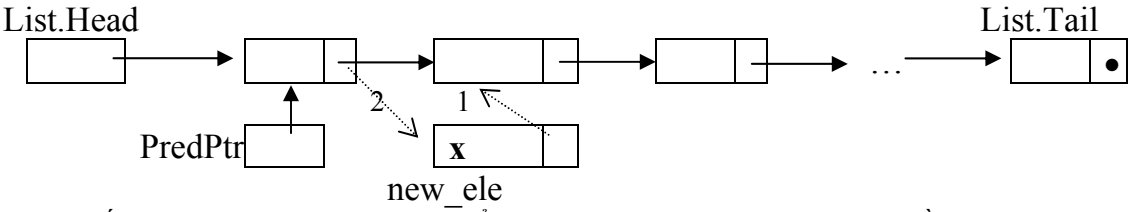
TraverseLL*(List)*
. CurrPtr = List.Head;
. Trong khi chưa hết DSLK thực hiện:
{ XửLý nút được trỏ bởi CurrPtr;
CurrPtr = CurrPtr->Next; // chuyển đến nút kế tiếp
}

- ***Cài đặt***

```
int TraverseLL(LL  List)
{ NodePointer CurrPtr = List.Head;
        if (EmptyLL(List)) return 0;
        else  { while (CurrPtr != NULL)  // hoặc   while (CurrPtr)
                  {   XửLý (CurrPtr);
                      CurrPtr = CurrPtr->Next;
                  }
                  return 1;
                }
}

void  XửLý(NodePointer CurrPtr)
{       // Xử  lý nút CurrPtr  tùy theo từng yêu cầu cụ thể. Có hai loại xử  lý:
        // 1. Xử  lý chỉ liên quan đến thông tin một nút
        // 2. Xử  lý liên quan đến thông tin của nhiều nút của DSLK
        return ;
}
```

- ***Thêm một phần tử mới vào DS***

**** Thêm một phần tử vào sau một nút được trỏ bởi con trỏ PredPtr***
(qui ước: nếu PredPtr == NULL thì chèn x vào đầu DSLK)



Áp dụng thao tác cơ bản trên, để cho gọn trong việc trình bày các phần sau, ta xây dựng thêm các thao tác sau:

- ***Thuật toán:*** *Thêm một nút new_ele vào sau một nút được trỏ bởi PredPtr*

InsertNodeAfterLL(&*List, new_ele, PredPtr)*
. if (PredPtr)
{ new_ele->Next = PredPtr->next;
PredPtr->Next = new_ele;
}
else { new_ele->Next = List.Head; // chèn new_ele vào đầu List
List.Head = new_ele;
}
// Nếu chèn new_ele vào cuối DS thì cần cập nhật lại đuôi của List
. if (PredPtr == List.Tail) List.Tail = new_ele;

- ***Cài đặt***

```
void InsertNodeAfterLL(LL  &List, NodePointer  new_ele, NodePointer PredPtr)
{       if (PredPtr)
                { new_ele->Next = PredPtr->next;
                  PredPtr->Next = new_ele;
                }
        else    { new_ele->Next = List.Head;
                  List.Head = new_ele;
                }
        if (PredPtr == List.Tail) List.Tail = new_ele;
        return  ;
}
```

- ***Thuật toán:*** *chèn thêm phần tử x vào sau một nút được trỏ bởi PredPtr. Hàm này trả về địa chỉ nút mới thêm vào,* nếu đủ vùng nhớ cấp phát cho nó; ngược lại, nó sẽ *trả trị NULL.*

NodePointer ***InsertElementAfterLL*** *(&List, x, PredPtr)*
. if ((new_ele = CreateNode (x)) == NULL) return NULL;
. Thêm nút new_ele vào sau nút được trỏ bởi PredPtr; Trả về new_ele;

- ***Cài đặt***

NodePointer ***InsertElementAfterLL*** *(LL &List, ElementType x, NodePointer PredPtr)*

```
{ NodePointer  new_ele;
        if  (! (new_ele = CreateNode (x))  return NULL;
        InsertNodeAfterLL (List, new_ele, PredPtr);
        return (new_ele);
}
```

* *Thêm một phần tử vào cuối một DSLK*

- ***Thuật toán:*** *Thêm một nút new_ele vào cuối DSLK List*
  InsertNodeTailLL*(&List, new_ele)*

  . Thêm nút new_ele vào sau nút được trỏ bởi List.Tail.

- ***Cài đặt***

```
void InsertNodeTailLL(LL &List, NodePointer new_ele)
{
        InsertNodeAfterLL (List, new_ele, List.Tail);
        return ;
}
```

- ***Thuật toán:*** *Thêm phần tử x vào cuối List*
  NodePointer ***InsertElementTailLL*** *(&List, x)*

  . Thêm phần tử x vào sau nút được trỏ bởi List.Tail.

- ***Cài đặt***

```
NodePointer InsertElementTailLL (LL  &List, ElementType x)
{
        return (InsertElementAfterLL (List, x, List.Tail));
}
```

* *Thêm một phần tử vào đầu một DSLK*

- ***Thuật toán:*** *Thêm một nút new_ele vào đầu DSLK List*
  InsertNodeHeadLL*(&List, new_ele)*

  . Thêm nút new_ele vào đầu List (hay sau nút được trỏ bởi NULL).

- ***Cài đặt***

```
void InsertNodeHeadLL(LL  &List, NodePointer new_ele)
{
        InsertNodeAfterLL (List, new_ele, NULL);
        return ;
}
```

- ***Thuật toán:*** *Thêm phần tử x vào đầu List*
  NodePointer ***InsertElementHeadLL*** *(&List, x)*

  . Thêm phần tử x vào đầu List (hay sau nút được trỏ bởi NULL).

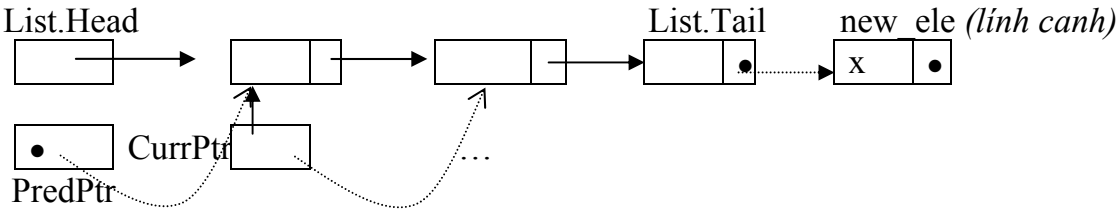
- ***Cài đặt***

```
NodePointer InsertElementHeadLL (LL  &List, ElementType x)
{
        return (InsertElementAfterLL (List, x, NULL));
```

}

- ***Tìm kiếm một phần tử trên DSLK***

Tìm một phần tử *x* trong DSLK *List*. Nếu *tìm thấy* thì, thông qua đối cuối của hàm, *trả về địa chỉ PredPtr của nút đứng trước nút tìm thấy đầu tiên*. Nếu *nút tìm thấy là nút đầu* của *List* thì *trả về* con trỏ *NULL*. Để *tăng tốc độ tìm kiếm* (bằng cách giảm số lần so sánh trong biểu thức điều kiện của vòng lặp), ta đặt *thêm lính canh* ở cuối List.



- ***Thuật toán tìm kiếm tuyến tính (có lính canh) trên dãy chưa được sắp:***

Boolean ***SearchLinearLL****(List, x, &PredPtr)*
. Chèn nút mới new_ele chứa x vào cuối List (đóng vai trò lính canh)
. PredPtr = NULL; CurrPtr = List.Head; // PredPtr đứng kề trước CurrPtr
. Trong khi (CurrPtr->Data ≠ x) thực hiện
{ PredPtr = CurrPtr; CurrPtr = CurrPtr->Next;
}
. if (CurrPtr ≠ new_ele) Thấy = True; // Thông báo thấy x;
else Thấy = False; // Thông báo không thấy x;
. Xóa nút (new_ele) đứng sau nút được trỏ bởi List.Tail;
. Trả về trị Thấy;

- ***Cài đặt***

```
int SearchLinearLL(LL List, ElementType x, NodePointer &PredPtr)
{ NodePointer  CurrPtr = List.Head,  OldTail= List.Tail,
               new_ele = InsertElementTailLL(List, x);
      PredPtr = NULL;
  int Thấy;
      while (SoSánh(CurrPtr->Data, x) != 0)
        { PredPtr = CurrPtr ;  CurrPtr = CurrPtr->Next;
        }
       if (CurrPtr != new_ele) Thấy = 1;          // thấy thật sự
      else Thấy = 0;                              // thấy giả hay không thấy !
      RemoveAfterLL(List, OldTail, x);  // xóa new_ele;
      return Thấy;
```

}

- ***Thuật toán tìm kiếm tuyến tính (có lính canh) trên dãy được sắp** (tăng)**:***

int ***SearchLinearOrderLL****(List, x, &PredPtr)*

. Chèn nút mới new_ele chứa x vào cuối List (đóng vai trò lính canh)

. PredPtr = NULL; CurrPtr = List.Head;

. Trong khi (CurrPtr->Data < x) thực hiện

{ PredPtr = CurrPtr ; CurrPtr = CurrPtr->Next;

}

. if ((CurrPtr ≠ new_ele) and (CurrPtr->Data ≡ x)) Thấy = True; // thấy x;

else Thấy = False; // không thấy x;

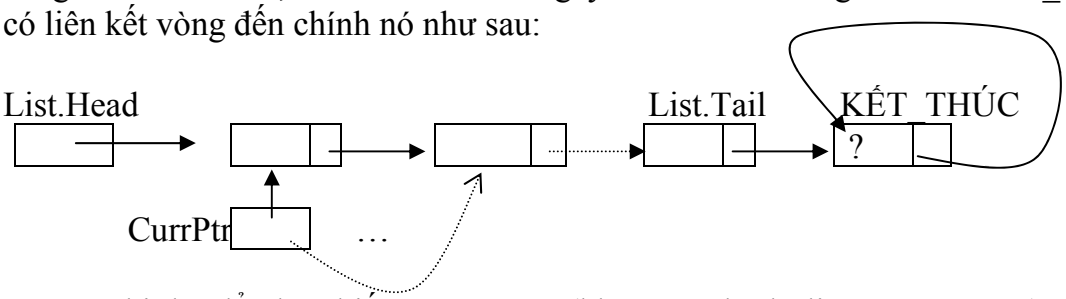
. Xóa nút (new_ele) đứng sau nút được trỏ bởi List.Tail;

. Trả về trị Thấy;

- ***Cài đặt***

```
int SearchLinearOrderLL(LL List, ElementType x, NodePointer &PredPtr)
{ NodePointer  CurrPtr = List.Head,  OldTail = List.Tail,
        new_ele = InsertElementTailLL(List, x);
        PredPtr = NULL;
  int Thấy;
        while (SoSánh(CurrPtr->Data, x) < 0)
          { PredPtr = CurrPtr;
            CurrPtr = CurrPtr->Next;
          }
         if ((CurrPtr != new_ele) && SoSánh(CurrPtr->Data, x) == 0) Thấy = 1;
        else Thấy = 0;
        RemoveAfterLL(List, OldTail, x);          // xóa new_ele;
        return Thấy;
}
```

Có một cách cài đặt khác cho DSLK đơn là: thay vì nhận biết hết DSLK bằng con trỏ NULL, ta có thể tạo mới ngay từ đầu một nút gọi là nút KẾT_THÚC có liên kết vòng đến chính nó như sau:

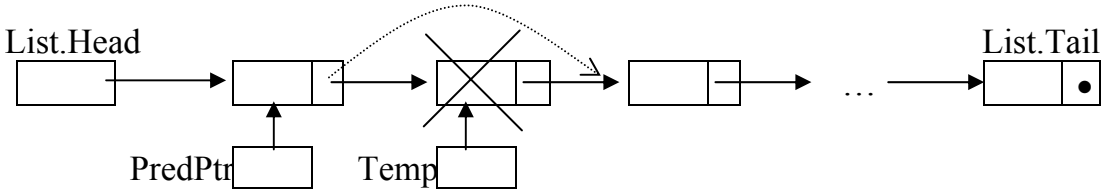


Khi đó, để nhận biết nút CurrPtr (không xử lý dữ liệu của nút này) có phải là nút kết thức hay không, ta dùng điều kiện *(CurrPtr->Next != CurrPtr)* thay cho *(CurrPtr != NULL)* trong biểu thức điều kiện để kết thúc vòng lặp *while*. Trong nhiều trường hợp, nút kết thúc này được sử dụng như nút lính canh để tăng tốc độ thực hiện của các thuật toán cần dùng lính canh ở cuối. Hãy viết lại các thuật toán cơ bản trên DSLK đơn được cài đặt theo cách này (*bài tập*).

- ***Xóa một phần tử khỏi DSLK***

* *Xóa một nút sau một nút được trỏ bởi con trỏ PredPtr*

(qui ước: nếu PredPtr == NULL thì xóa nút đầu)



- ***Thuật toán***

RemoveAfterLL(&*List, PredPtr, &****x)***

. if (PredPtr)
{ Temp = PredPtr->Next;
if (Temp) PredPtr->Next = Temp->Next;
}
else // xóa nút đầu
{ Temp = List.Head;
List.Head = Temp->Next;
}
. if (Temp == List.Tail) List.Tail = PredPtr;//nếu xóa đuôi, cần cập nhật lại đuôi
. x = Temp->Data; delete Temp;

- ***Cài đặt***

```
int RemoveAfterLL(LL  &List, NodePointer PredPtr, ElementType &x)
{ NodePointer Temp;
    if (EmptyLL(List))
        { cout << "\nDS rỗng !";          // không có gì để xoá !
          return 0;
        }
    if (PredPtr)
        { Temp = PredPtr->Next;
          if (Temp == NULL) return 0;     // không thể xóa nút sau nút cuối !
          else PredPtr->Next = Temp->Next;
        }
    else  { Temp = List.Head;                       // xóa nút đầu
          List.Head = Temp->Next;
        }
    if (Temp == List.Tail) List.Tail = PredPtr; //nếu xóa đuôi, cần cập nhật lại đuôi
    Gán(x, Temp->Data);
    delete Temp;
```

```
        return 1; // xóa thành công
}
```

* *Xóa nút đầu của DSLK*

- ***Thuật toán:*** *Xóa nút đầu của DSLK List*
  int ***RemoveHeadLL(&****List, &****x)***
  . Xóa nút đầu (hay sau nút được trỏ bởi NULL) của List.

- ***Cài đặt***

```
int RemoveHeadLL(LL  &List, ElementType &x)
{
        return RemoveAfterLL (List, NULL, x);
}
```

* *Xóa một phần tử x khỏi DSLK*

- ***Thuật toán:***
  int ***RemoveElementLL(&****List,* ***x)***

. Tìm *x* trong *List;*
. Nếu thấy thì:
- Trả về biến con trỏ *PredPtr* chỉ đến nút đứng trước nút tìm thấy;
- Xóa nút đứng sau nút được trỏ bởi *PredPtr*.

Ngược lại thì kết thúc;

- ***Cài đặt***

```
int RemoveElementLL(LL  &List, ElementType x)
{ NodePointer  PredPtr;
        if (!SearchLinearLL(List, x, PredPtr)) return 0;
        else return RemoveAfterLL (List, x, PredPtr);
}
```

c. Sắp xếp trên kiểu DSLK đơn

Có hai cách chính thực hiện các thuật toán sắp xếp trên DSLK:

* *Cách 1*: *Hoán vị nội dung dữ liệu* (trường Data) của các nút trên DSLK tương tự như cách sắp xếp trên mảng đã trình bày trong chương trước. Điểm *khác biệt* là việc *truy xuất* đến các phần tử trên DSLK sẽ *theo trường liên kết Next thay vì theo chỉ số* như trên mảng. Với cách tiếp cận này, nếu *kích thước trường dữ liệu lớn* thì *chi phí cho việc hoán vị các cặp phần tử sẽ rất lớn* (do đó, tốc độ thực hiện các thuật toán sắp xếp sẽ rất chậm). Vả lại, cách làm như vậy sẽ không tận dụng được ưu điểm linh hoạt của DSLK động trong các thao tác chèn và xóa (chẳng hạn đối với thuật toán sắp xếp chèn trực tiếp).

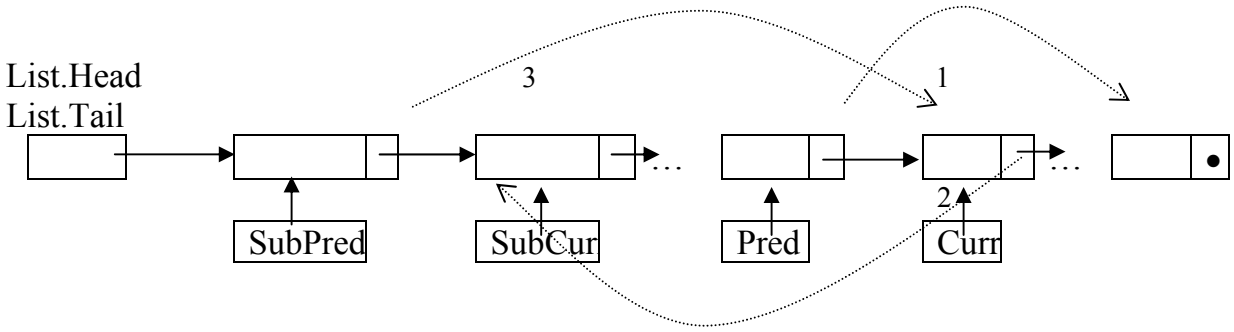
* *Cách 2*: Thay vì hoán vị nội dung dữ liệu của các nút, ta *chỉ thay đổi* thích hợp *các trường liên kết* Next giữa những nút để được thứ tự mong muốn. *Kích thước của trường liên kết*: *không phụ thuộc* vào bản thân nội dung *dữ liệu* của các phần tử, *cố định trong mỗi môi trường* 16 bits hay 32 bits và thường là *khá nhỏ* so với kích thước của trường dữ liệu trong các ứng dụng lớn trên thực tế. Tuy

nhiên, các *thao tác trên trường liên kết này thường phức tạp hơn* trên trường dữ liệu.

Trong phần này, ta sẽ xét một số thuật toán *sắp xếp có tận dụng các ưu thế của DSLK động*.

- *Sắp xếp chèn trực tiếp trên DSLK*

Trước hết, ta minh họa thuật toán *sắp xếp chèn trực tiếp* một dãy các đối tượng *được cài đặt bằng DSLK động* thông qua kiểu con trỏ. Lưu ý rằng, *tận dụng ưu điểm liên kết động* của con trỏ *trong thao tác chèn*, *thay vì phải dời chỗ* (*chi phí dời chỗ phụ thuộc vào chiều dài của dãy con* và do đó chiếm rất nhiều thời gian) các dãy con nhằm tìm vị trí thích hợp để chèn phần tử mới vào dãy con cũ đã được sắp, ta *chỉ phải thay đổi liên kết của không quá ba nút* (*chi phí hằng*, không phụ thuộc vào chiều dài dãy con, do đó sẽ rút ngắn thời gian đáng kể cho những phép hoán vị hay dời chỗ các phần tử).



- ***Thuật toán***

SắpXếpChènLL*(&List)*

- Bước 1: Pred = List.Head; // DS từ đầu đến *PredPtr đã được sắp*
  Curr = Pred->Next; // Con trỏ Curr kề sau Pred
- Bước 2: Trong khi (Curr ≠ NULL) thực hiện:
  . Bước 2.1: SubCurr = List.Head; // Bắt đầu tìm từ List.Head
  SubPred = NULL; // nút đứng trước SubCurr
  // Tìm vị trí *SubPred* thích hợp để chèn Curr sau
  // SubPred, dùng Curr làm lính canh
  . Bước 2.2:Trong khi (SubCurr->Data<Curr->Data) thực hiện:
  { SubPred = SubCurr;
  SubCurr = SubCurr->Next;
  }
  . Bước 2.3: if (SubCurr ≠ Curr)
  { Pred->Next = Curr->Next;
  Chèn nút Curr sau SubPred;
  }

else Pred = Curr; // Curr đã đặt đúng vị trí

. Bước 2.4: Curr = Pred->Next;

- ***Cài đặt***

```
void SắpXếpChènLL(LL &List)
{ NodePointer  Pred = List.Head,       // DS con từ  List.Head đến PredPtr đã được sắp
               Curr = Pred->Next,     // Curr là con trỏ đứng sau Pred
               SubCurr, SubPred;
// SubPred là nút kề trước SubCurr, dùng để tìm vị trí để chèn Curr trong dãy con
        while (Curr)
        { SubPred = NULL; SubCurr = List.Head; // Bắt đầu tìm từ  List.Head
          while (SoSánh(SubCurr->Data, Curr->Data) < 0)
          { SubPred = SubCurr;  SubCurr = SubCurr->Next;
          }
          if (SubCurr != Curr)   // Chèn Curr sau SubPred
                { Pred->Next = Curr->Next;
                  InsertNodeAfterLL(List, Curr, SubPred);
        }
          else    Pred = Curr;
          Curr = Pred->Next;
        }
        return ;
}
```

Sau đây, ta sẽ xét thêm một số thuật toán sắp xếp khác được cài đặt bằng DSLK động thể hiện một cách đơn giản và rõ hơn bản chất của phương pháp và tỏ ra khá hiệu qủa: *Quick sort, Natural Merge sort (sắp trộn tự nhiên)* và *Radix sort.*

- <u>*Phương pháp QuickSort trên DSLK*</u>

  Do đặc điểm của DSLK đơn, để *giảm chi phí tìm kiếm*, ta nên *chọn mốc là phần tử ở đầu* DSLK.

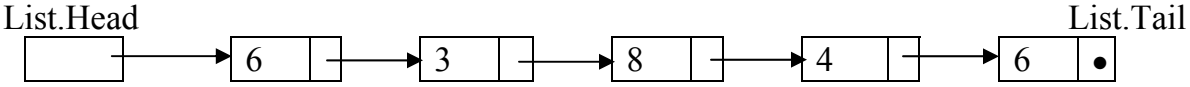
- ***Thuật toán***

QuickSortLL*(&List)*

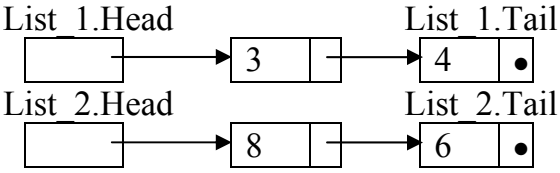
- Bước 1: Chọn phần tử đầu List.Head làm mốc *g*. Loại g khỏi List.
- Bước 2:Tách List thành hai DSLK con List_1 (gồm những phần tử có trị nhỏ hơn *g*) và List_2 (gồm những phần tử có trị lớn hơn hoặc bằng hơn *g*)
- Bước 3: if (List_1 ≠ NULL) *QuickSortLL* (List_1);
  if (List_2 ≠ NULL) *QuickSortLL* (List_2);
- Bước 4: Nối List_1, *g,* List_2 theo trình tự đó thành List được sắp.

Chú ý rằng, khi tách List thành hai DSLK con List_1 và List_2, ta không sử dụng thêm bộ nhớ phụ (mà phụ thuộc vào chiều dài danh sách).

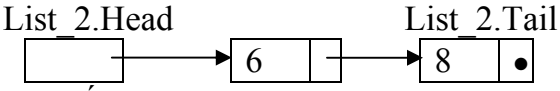
* *Ví dụ* Sắp xếp tăng DSLK sau:



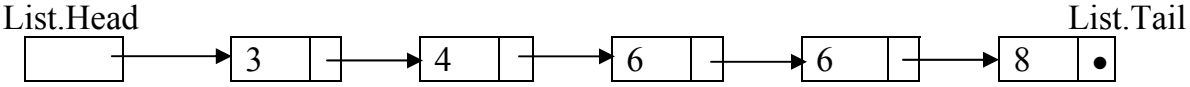
. Chọn nút đầu tiên làm mốc: g = 6. Tách List thành hai DSLK con:



. Với List_2, chọn g = 8. Tách List_2 thành hai DSLK con. Sau đó nối lại, ta được:



. Nối List_1, g = 6 và List_2, ta được List được sắp:



- ***Cài đặt***

```
void QuickSortLL(LL &List)
{ NodePointer g, Temp;
  LL  List_1, List_2;
      if (List.Head == List.Tail) return; // List được sắp nếu  nó: rỗng hay có 1 phần tử
      g = List.Head;
      List.Head = List.Head->Next;      // tách g ra khỏi List
      List_1 = CreateEmptyLL();
      List_2 = CreateEmptyLL();
      while (!EmptyLL(List))
      { Temp = List.Head;
        List.Head = List.Head->Next; Temp->Next = NULL;
        if (SoSánh(Temp->Data, g->Data) < 0) InsertNodeTailLL(List_1,Temp);
        else InsertNodeTailLL(List_2,Temp);
      }
      QuickSortLL(List_1);
      QuickSortLL(List_2);
      // Nối g sau List_1
      if (EmptyLL(List_1)) List.Head = g;
      else {  List.Head = List_1.Head;
              List_1.Tail->Next = g;
           }
      g->Next = List_2; // Nối List_2 sau g
```

```
        if ((EmptyLL(List_2)) List.Tail = g; //Cập nhật lại đuôi của List
        else List.Tail = List_2.Tail;
        return;
}
```

- *Phương pháp NaturalMergeSort trên DSLK*

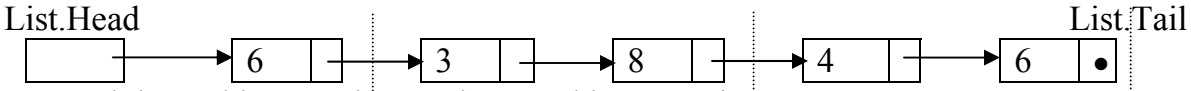
Khi cài đặt dãy cần sắp bằng phương pháp *trộn tự nhiên* trên DSLK đơn, bằng cách *thay đổi các liên kết* cho phù hợp ta có dãy được sắp mà *không cần phải dùng dãy phụ lớn* (kích thước *phụ thuộc vào cỡ dãy*) *như đã làm trên mảng*.

- ***Thuật toán***

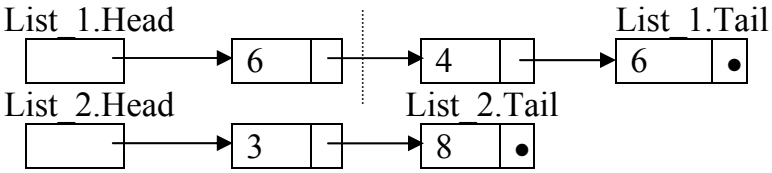
NaturalMergeSortLL *(&List)*

- Bước 1: Phân phối luân phiên từng đường chạy của List vào hai DSLK List_1 và List_2;
- Bước 2: if (List_1 ≠ NULL) *NaturalMergeSortLL* (List_1);
  if (List_2 ≠ NULL) *NaturalMergeSortLL* (List_2);
- Bước 3: Trộn List_1 và List_2 đã sắp để có List được sắp;

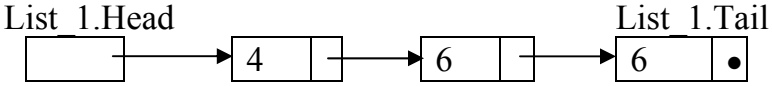
* *Ví dụ* Sắp xếp tăng DSLK sau:



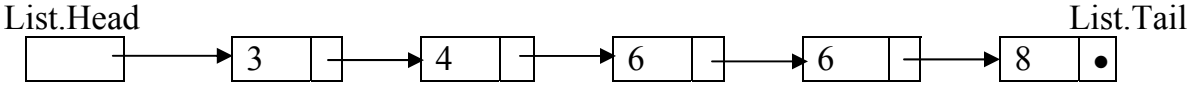
. Tách luân phiên các đường chạy tự nhiên của List vào 2 DSLK con:



. Lại tách luân phiên các đường chạy tự nhiên của List_1 vào 2 DSLK con, rồi sau đó trộn lại, ta được List_1 tăng:



. Trộn List_1 và List_2, ta được List tăng:



- ***Cài đặt***

```
void NaturalMergeSortLL (LL &List)
{ LL List_1, List_2;
        if (List.Head == List.Tail) return; // List được sắp nếu nó: rỗng hay có 1 phần tử
        List_1 = CreateEmptyLL(); List_2 = CreateEmptyLL();
        // Phân phối các đường chạy của List vào List_1 và List_2
        DistributeLL(List, List_1, List_2);
        if (Empty(List_2) { List = List_1; return; }
        NaturalMergeSortLL (List_1); NaturalMergeSortLL (List_2);
```

```
        // Trộn hai DSLK đã sắp List_1 và List_2  thành List
        MergeLL(List_1, List_2, List);
        return;
}

void MergeLL(LL &List_1, LL &List_2, LL &List)
{ NodePointer Temp;
        while (!EmptyLL(List_1) &&  !EmptyLL(List_2))
        { if (SoSánh(List_1.Head->Data, List_2.Head->Data) <= 0)
                { Temp = List_1.Head;              // Tách Temp ra khỏi List_1
                  List_1.Head = List_1.Head->Next;
                }
          else  { Temp = List_2.Head;    // Tách Temp ra khỏi List_2
                  List_2.Head = List_2.Head->Next;
                }
          Temp->Next = NULL;
          InsertNodeTailLL(List, Temp);
        }
   LL   ListCònLại = List_1;
        if (EmptyLL(List_1))  ListCònLại = List_2;
        if (!EmptyLL(ListCònLại))
                { List.Tail->Next = ListCònLại.Head;
                  List.Tail = ListCònLại.Tail;
                }
        return ;
}

void DistributeLL(LL &List, LL &List_1, LL &List_2)
{ NodePointer Temp;
        do
        { Temp = List.Head;                              // Tách Temp ra khỏi List
          List.Head = List.Head->Next ;
          Temp->Next = NULL;
          InsertNodeTailLL(List_1, Temp);
        } while (List.Head && (Sosánh(Temp->Data, List.Head->Data) <= 0));
        if (List.Head) DistributeLL(List, List_2, List_1);
        else List.Tail = NULL; //Cập nhật lại đuôi rỗng cho List, chuẩn bị cho phép trộn
        return ;
}
```

<u>*Chú ý*</u>: Trong vòng lặp của thủ tục *DistributeLL* trên đây để tìm và đưa một đường chạy tự nhiên vào một DSLK con, ta thực hiện *thừa các phép nối thêm những nút của List vào đuôi của DSLK con* (*chi phí* thực hiện *các phép nối thêm* này *phụ thuộc vào độ dài mỗi đường chạy*). Ta có thể viết thêm các module con: *tìm một đường chạy tự nhiên* từ vị trí hiện hành (chỉ có phép so sánh) và *phép nối một đường chạy đó vào đuôi của DSLK con* tương ứng. Khi đó *chi phí cho phép nối thêm này là hằng, không phụ thuộc vào độ dài mỗi đường chạy* (*tại sao* ? B*ài tập*).

- <u>*Phương pháp RadixSort trên DSLK*</u>

Khi cài đặt thuật toán RadixSort trên cấu trúc dữ liệu mảng, ta lãng phí bộ nhớ quá nhiều. Các cài đặt thuật toán này trên DSLK động sẽ trình bày sau đây sẽ khắc phục được nhược điểm trên. Giả sử ta cần sắp (tăng) một dãy số nguyên mà số chữ số tối đa của chúng là *m*.

- ***Thuật toán***

RadixSortLL *(&List, m)* // *m* là số ký số tối đa của dãy số cần sắp

- Bước 1: k = 0; // k = 0: hàng đơn vị, k = 1: hàng chục, …
- Bước 2: .Khởi tạo 10 DSLK (lô) rỗng: $B_0, ..., B_9$;
  .Trong khi (List ≠ rỗng) thực hiện:
  { Temp = List.Head; List.Head = List.Head->Next;
  Temp->Next = NULL; //Tách nút đầu Temp ra khỏi List
  Chèn nút Temp vào cuối DSLK B_i;
  // với i là chữ số thứ i của Temp->Data;
  }
- Bước 3: Nối lần lượt các DSLK $B_0, ..., B_9$ thành List;
- Bước 4: k = k +1;
  if (k < m) Quay lại bước 2;
  else Dừng;

- ***Cài đặt***

```
#define MAX_LO 10
void  RadixSortLL (LL &List, int m)
{ LL B[MAX_LO];
  NodePointer Temp;
  int i, k;
        if (List.Head == List.Tail) return ;// List được sắp nếu  nó: rỗng hay có 1 phần tử
        for (k = 0;  k < m; k++)
        { for (i = 0;  i < MAX_LO;  i++)  CreateEmptyLL(B[i]);
          while (!EmptyLL(List))
          { Temp = List.Head; List.Head = List.Head->Next;
            Temp->Next = NULL; //Tách nút đầu Temp ra khỏi List
             InsertNodeTailLL(B[GetDigit(Temp->Data, k)], Temp);
          }
          List = B[0];
          for (i = 1;  i < MAX_LO;  i++) AppendList(List,B[i]); // Nối B[i] vào cuối List
        }
        return ;
}

void AppendList(LL &List, LL List_1)              // Nối List_1 vào cuối List
{       if (Empty(List_1)) return;
        if (Empty(List)) List = List_1;
        else
              {  List.Tail->Next = List_1.Head;
                 List.Tail = List_1.Tail;
```

```
            }
        return ;
}

int GetDigit(unsigned long N, int k)                    // Lấy chữ số thứ  k của số nguyên N
{
        return ((unsigned long)(N/pow(10,k)) % 10);      // pow (x, y) ≡ x^y
}
```

III.3.2. Vài ứng dụng của DSLK đơn

III.3.2.1. Ngăn xếp

a. Định nghĩa

Ngăn xếp (*stack*) là kiểu dữ liệu tuyến tính nhằm biểu diễn các đối tượng được xử lý theo kiểu *"vào sau ra trước"* (*LIFO*: Last In, First Out). Ta có thể *dùng danh sách* để biểu diễn ngăn xếp, các phép toán *thêm vào* và *lấy ra* được thực hiện *cùng ở một đầu* danh sách *(*gọi là *đỉnh* của ngăn xếp).

Ta cũng có thể định nghĩa stack là một kiểu dữ liệu trừu tượng tuyến tính, trong đó có hai thao tác chính:

- *Push*(O): thêm một đối tượng O vào đầu stack;
- *Pop*(): lấy ra một đối tượng ở đầu stack và trả về trị của nó, nếu stack rỗng sẽ gặp lỗi;

và thêm hai thao tác phụ trợ khác:

- *EmptyStack*(): kiểm tra xem stack có rỗng hay không;
- *Top*(): Trả về trị của phần tử ở đầu stack mà không loại nó khỏi stack, nếu stack rỗng sẽ gặp lỗi.

* *Ví dụ*: Ta có thể dùng ngăn xếp để cài đặt thuật toán đổi một số nguyên dương từ cơ số 10 sang cơ số 2 (bài tập).

Ta có thể dùng *mảng* hay *DSLK động* để *biểu diễn stack*.

b. Cài đặt ngăn xếp bằng mảng

- ***Cài đặt cấu trúc dữ liệu***

Ta còn có thể cài đặt ngăn xếp *S* bằng *mảng* 1 chiều có *kích thước tối đa* là *N,* các phần tử của nó được đánh số bắt đầu từ 0 (đến N-1), phần tử ở đỉnh stack có chỉ số là *t*. Dựa trên cơ sở đó, trong C++, *stack* có thể được quản lý thông qua cấu trúc sau:

```
typedef struct { ElementType mang[N];
                 int  t ; // chỉ số của đỉnh stack
               } StackType;
StackType S;
```

S.mang[0]	S.mang[1]	…	S.mang[t-1]	t	
X	y		Z		

- ***Các phép toán cơ bản trên stack***

```
StackType  CreateEmptyStack()
{ StackType  S;
        S.t == 0; return  S;
}

int  EmptyStack(StackType  S)
{       return (S.t == 0);
}
```

Do kích thước của mảng cố định, *trước khi chèn* ta phải *kiểm tra ngăn xếp đã đầy* hay chưa thông qua hàm *FullStack* sau đây.

```
int  FullStack(StackType  S)
{       return (S.t >= N);
}

int Push(StackType  &S, ElementType x)
{       if (FullStack(S)) return 0; // Stack đầy, chèn không thành công
        else    { S.mang[t++] = x; return 1;
                }
}

int Pop (StackType  &S, ElementType &x)
{       if (EmptyStack(S)) return 0; // Stack rỗng, không lấy được phần tử  ở đỉnh S
        else    { x = S.mang[--t]; return 1;
                }
}

int Top (StackType  S, ElementType &x)
{       if (EmptyStack(S)) return 0; // Stack rỗng, không xem được phần tử  ở đỉnh S
        else    { x = S.mang[t-1]; return 1;
                }
}
```

- *Nhận xét*:

- Các thao tác trên đều *đơn giản, hiệu quả* và *có chi phí hằng số* O(1)
- *Hạn chế* của cách cài đặt này: kích thước của stack bị giới hạn và kém linh động, do đó việc sử dụng bộ nhớ kém hiệu quả (*thiếu hay lãng phí bộ nhớ*).

Sau đây, ta sẽ tập trung khảo sát cách cài đặt ngăn xếp bằng DSLK động.

c. Cài đặt ngăn xếp bằng DSLK động

- ***Cài đặt.***

Ta có thể cài đặt ngăn xếp *bằng danh sách liên kết động (tương tự như DSLK đơn, chỉ khác* là *không lưu đến nút cuối* hay *đáy của ngăn xếp)* như sau:

```
typedef    ....   ElementType;                    // Kiểu dữ  liệu của nút
typedef struct node { ElementType  Data;
                      struct node *Next;
                    } NodeType;
typedef  NodeType  *NodePointer;
NodePointer  Stack;
```

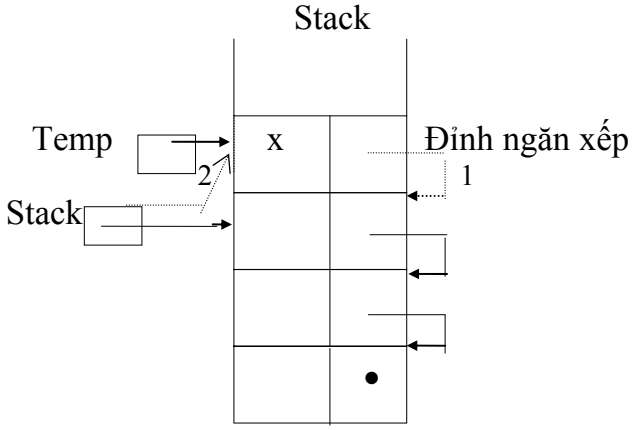
- ***Các phép toán cơ bản trên stack***

Các thao tác khởi tạo một stack rỗng và kiểm tra xem một stack cho trước có rỗng hay không tương tự như DSLK đơn. Ta chỉ chú trọng đến hai thao tác đặc trưng của ngăn xếp là lấy ra *Pop* và thêm vào *Push* ở đỉnh ngăn xếp.

Gọi Stack là con trỏ chỉ đến phần tử ở đỉnh của ngăn xếp.

* *Thao tác **Push** đẩy một mục dữ liệu x vào đỉnh ngăn xếp*

Thao tác *Push* tương tự thao tác *InsertElementHeadLL,* nếu ta quản lý thêm nút ở đáy *stack.*

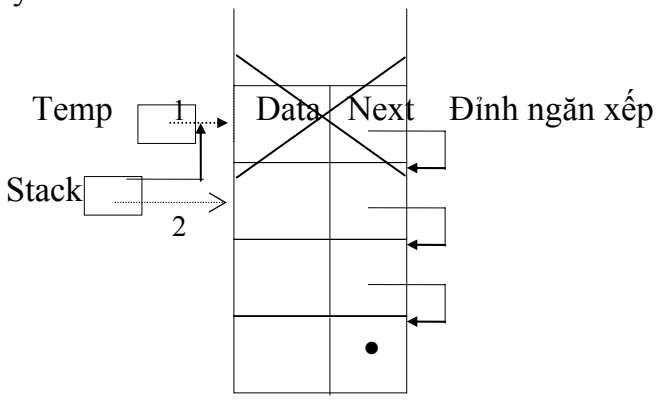


Hoặc ta có thể viết trực tiếp như sau:

```
int Push(NodePointer &Stack, ElementType x)
{ NodePointer Temp;
        if ((Temp = CreateNodeLL(x)) == NULL) return(0);
        else { Temp->Next  = Stack;
               Stack = Temp;
               return 1 ;
             }
}
```

* *Thao tác **Pop** lấy ra một phần tử ở đỉnh ngăn xếp*

Thao tác *Pop* tương tự thao tác *RemoveHeadLL,* nếu ta quản lý thêm nút ở đáy *stack.*



Ta có thể viết trực tiếp thao tác này như sau:

```
int Pop(NodePointer &Stack, ElementType &x)
{ NodePointer Temp;
        if (EmptyStack(Stack))
              {  cout << "\nNgăn xếp rỗng. Không thể lấy phần tử ở đỉnh ngăn xếp
!";
                 return 0;
              }
        else { Gan (x, Stack->Data);
                 Temp = Stack; Stack = Stack->Next;
                 delete Temp;
                 return 1;
              }
}
```

* *Thao tác **Top** xem một phần tử ở đỉnh ngăn xếp*

```
int Top(NodePointer Stack, ElementType &x)
{ NodePointer Temp;
        if (EmptyStack(Stack))
              {  cout << "\nNgăn xếp rỗng. Không thể xem phần tử  ở đỉnh ngăn xếp !";
                 return 0;
              }
        else { Gan (x, Stack->Data); return 1;
              }
}
```

d. Ứng dụng của ngăn xếp

Ngăn xếp có rất nhiều ứng dụng trong tin học: cài đặt phép *đệ qui*, khử đệ qui, *lưu vết* trong thuật toán *quay lui, vét cạn* hay *tìm kiếm theo chiều sâu*, trong

việc *chuyển đổi* giữa *các dạng kí pháp* khác nhau cũng như đánh giá các biểu thức chứa các toán tử không quá hai ngôi như biểu thức số học, lô-gic, …

Sau đây, ta dùng *ký pháp nghịch đảo Balan (*ký pháp *hậu tố RPN* - Reverse Polish Notation*)* để đánh giá các biểu thức số học. Một biểu thức số học *InfixeExp* thông thường được viết theo ký pháp *trung tố* (*toán tử đặt ở giữa hai* toán hạng). Ta sẽ ứng dụng ngăn xếp để: *chuyển InfixeExp sang dạng hậu tố SuffixeExp (toán tử đặt sau* các toán hạng*)* và *tính trị* của *SuffixeExp*.

* <u>*Ví dụ*</u>:

(1 + 5) * (8 - (4 - 1)) —Biến đổi→ 1 5 + 8 4 1 - - * —Đánh giá→ 30

(Ký pháp trung tố) (Ký pháp hậu tố)

Ta sẽ lần lượt xét hai thuật toán:

- Biến đổi biểu thức từ dạng kí pháp trung tố thành biểu thức dạng RPN.
- Đánh giá biểu thức số học dưới dạng RPN.

* ***Thuật toán chuyển biểu thức dạng trung tố sang dạng hậu tố RPN***

1. Khởi tạo ngăn xếp (dùng để *chứa các toán tử*) *S* rỗng;
2. Lặp lại các việc sau cho đến khi dấu kết thúc biểu thức được đọc:

. Đọc phần tử tiếp theo (hằng, biến, toán tử, '(', ')') trong biểu thức trung tố;

. Nếu phần tử là:

- Dấu '(': đẩy nó vào *S*;
- Dấu ')': hiển thị các phần tử của *S* cho đến khi dấu '(' (không hiển thị) được đọc;
- Toán tử:

| Nếu *S* rỗng: đẩy toán tử vào *S;* // (1)
| Ngược lại:
| | Nếu toán tử đó có độ ưu tiên cao hơn toán tử ở đỉnh *S* thì:
| | đẩy toán tử đó vào S;
| | Ngược lại: lấy ra và hiển thị toán tử ở đỉnh *S* ;
| | Quay lại (1);

- Toán hạng (hằng hoặc biến): Hiển thị nó;

3. Khi đạt đến dấu kết thúc biểu thức thì lấy ra và hiển thị các toán tử của *S* cho đến khi *S* rỗng;

(trong đó, ta xem dấu '(' có độ ưu tiên thấp hơn độ ưu tiên các toán tử +, -, *, /, %)

<u>*Ví dụ*</u>: Chuyển biểu thức 7*8-(2+3) sang dạng hậu tố.

Biểu thức kí pháp trung tố Stack *S* Hiển thị

7*8-(2+3) ____ 7

Lấy ra ↑

*8-(2+3) | * | 7

↑

8-(2+3) | * | 7 8

↑

-(2+3) | - | 7 8 *

↑ (ĐộƯuTiên[-] < ĐộƯuTiên[*]: lấy và hiển thị * ; *S* rỗng: đẩy – vào *S*)

(2+3) | (| - | 7 8 *

↑

2+3) | (| - | 7 8 * 2

↑

+3) | + | (| - | 7 8 * 2

↑ (ĐộƯuTiên[+] > ĐộƯuTiên[(]: đẩy + vào *S*)

3) | + | (| - | 7 8 * 2 3

↑

) | - | 7 8 * 2 3 +

↑

[Lấy ra + ( ]

Dấu kết thúc biểu thức, lấy - ra ↑ —— Kết quả ——→ 7 8 * 2 3 + -

* ***Thuật toán đánh giá biểu thức dạng RPN***

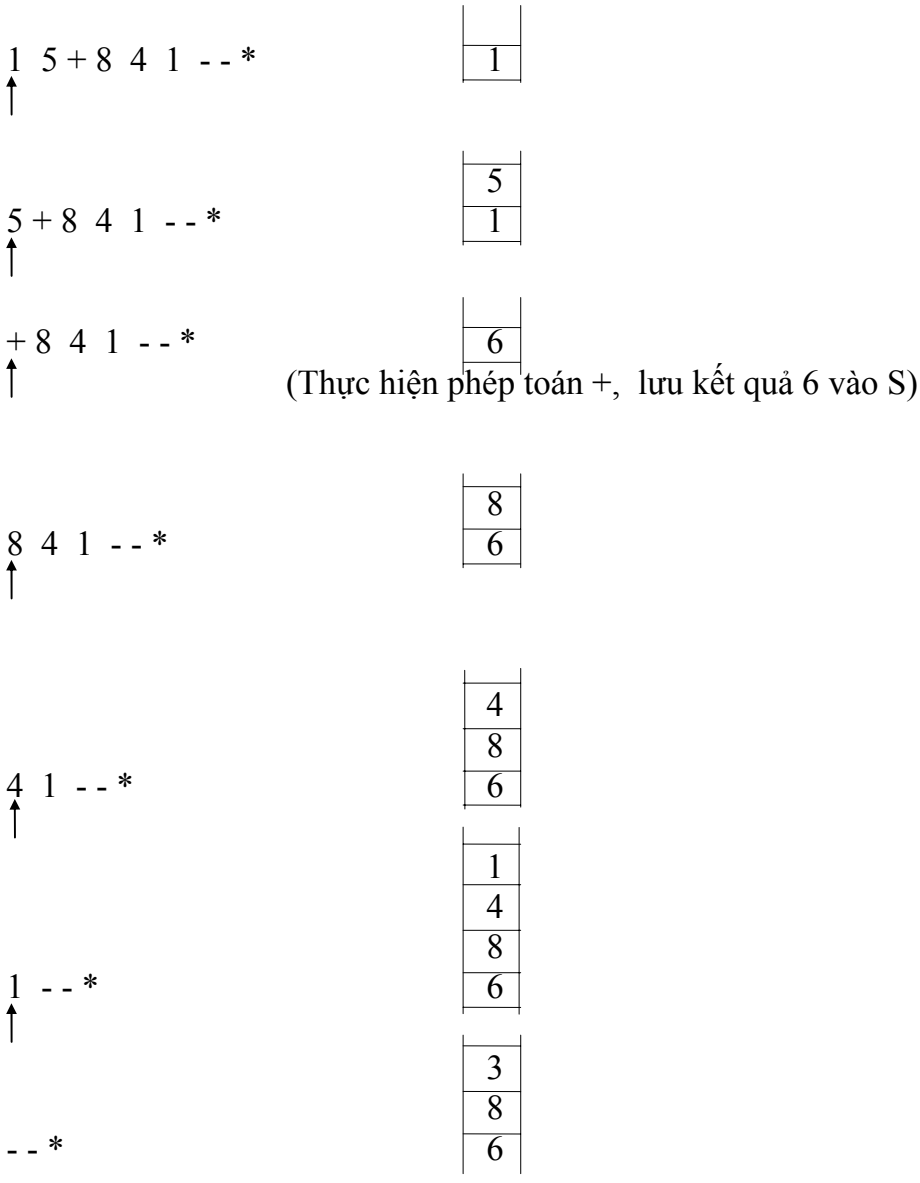
1. Khởi tạo ngăn xếp *S* rỗng;
2. Lặp lại các việc sau cho đến khi dấu kết thúc biểu thức được đọc:

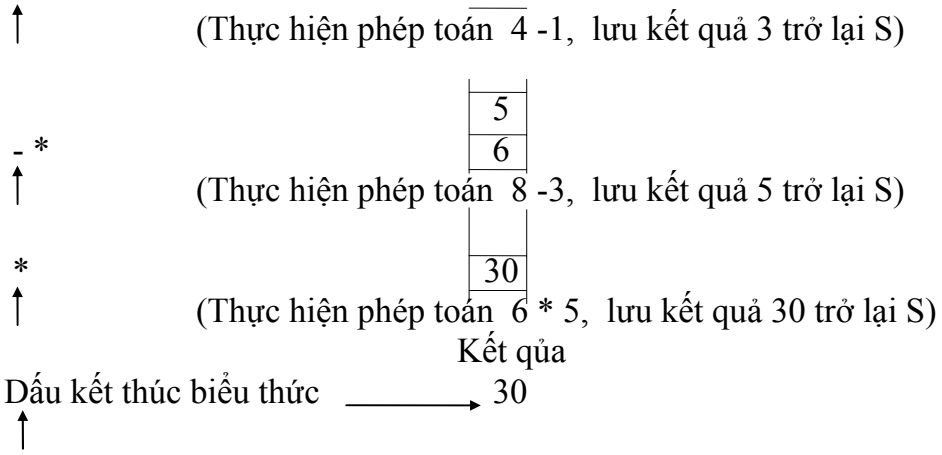
. Đọc phần tử (toán hạng, toán tử) tiếp theo trong biểu thức;
. Nếu phần tử là toán hạng: đẩy nó vào *S*;
Ngược lại: // phần tử là toán tử
- Lấy từ đỉnh *S* hai toán hạng;
- Áp dụng toán tử đó vào 2 toán hạng (theo thứ tự ngược);
- Đẩy kết qủa vừa tính trở lại *S*;

3. Khi gặp dấu kết thúc biểu thức, giá trị của biểu thức chính là giá trị ở đỉnh S;

Ví dụ: Tính giá trị của biểu thức hậu tố: 1 5 + 8 4 1 - - *

Biểu thức hậu tố Stack *S*





Chú ý rằng, trong các thuật toán không kiểm tra biểu thức đưa vào có đúng cú pháp hay không? Hãy bổ sung chức năng kiểm tra cú pháp cho các biểu thức (bài tập).

Ta có thể dùng ngăn xếp để khử đệ qui. Hãy khử đệ qui và viết lại dưới dạng lặp thuật toán *Quick Sort (bài tập)*. Chú ý, để *tiết kiệm bộ nhớ cho stack*, ta nên *lưu* vào ngăn xếp các *cặp chỉ số của dãy con nào dài hơn* !

III.3.2.2. Hàng đợi

a. <u>Định nghĩa</u>

Hàng đợi (*queue*) là kiểu dữ liệu tuyến tính nhằm biểu diễn các đối tượng được xử lý theo kiểu *"vào trước ra trước"* (*FIFO*: First In, First Out). Ta có thể *dùng danh sách* để biểu diễn hàng đợi, các phép toán *thêm vào* và *lấy ra* được thực hiện *ở hai đầu* khác nhau của danh sách.

Ta cũng có thể định nghĩa hàng đợi là một kiểu dữ liệu trừu tượng tuyến tính, trong đó các hai thao tác chính:

- *EnQueue*(O): *thêm* một đối tượng O *vào đuôi* hàng đợi;
- *DeQueue*(): *lấy ra* một đối tượng *ở đầu* hàng đợi và trả về trị của nó, nếu hàng đợi rỗng sẽ gặp lỗi;

và thêm hai thao tác phụ trợ khác:

- *EmptyQueue*(): kiểm tra xem hàng đợi có rỗng hay không;
- *Front* (): Trả về trị của phần tử ở đầu hàng đợi mà không loại nó khỏi hàng đợi, nếu hàng đợi rỗng sẽ gặp lỗi.

Ta có thể dùng *mảng vòng* hay *DSLK động* để biểu diễn hàng đợi.

b. <u>Cài đặt hàng đợi bằng mảng vòng</u>

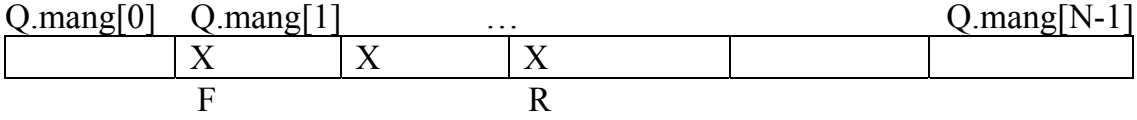
- ***Cài đặt cấu trúc dữ liệu***

Ta có thể biểu diễn hàng đợi *Q* bằng một *mảng* 1 chiều có kích thước tối đa là *N*. Để có thể sử dụng linh hoạt bộ nhớ mà mảng được cấp phát, ta tổ chức mảng

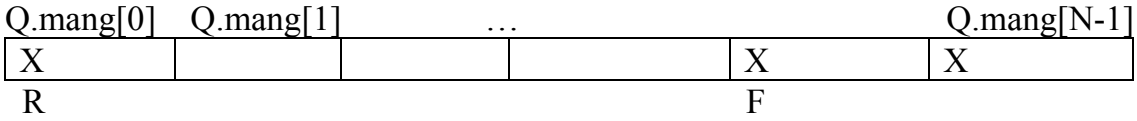
theo kiểu *xoay vòng* (nghĩa là *phần tử thứ N-1* được xem là *kề trước phần tử thứ 0*). Ngoài ra, ta còn lưu trữ *thêm hai chỉ số F* và R để *lưu vị trí* phần tử ở *đầu* và *đuôi* hàng đợi *Q*.

Trong C++, ta có thể quản lý hàng đợi thông qua mảng như sau:

```
typedef struct { ElementType mang[N];
                 int F, R ; // chỉ số của phần tử đầu và đuôi hàng đợi
               } QueueType;
QueueType Q;
```



Sau quá trình cập nhật (dãy các thao tác xóa, chèn), hàng đợi Q có thể "xoay vòng" như sau (X dùng để chỉ những vị trí chứa dữ liệu thật sự đang quan tâm trong hàng đợi):



- ***Các phép toán cơ bản***

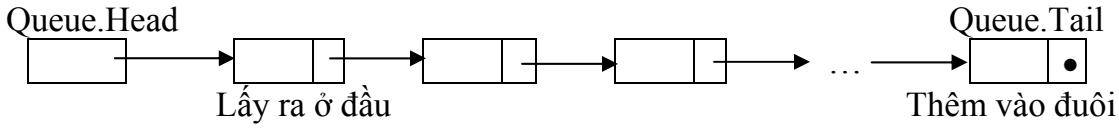
```
void CreateEmptyQueue (QueueType &Q)
{       Q.F = Q.R = -1; return ;
}

int EmptyQueue (QueueType Q)
{       return(Q.F == -1); // hoặc: return(Q.F == -1 && Q.R == -1);
}

int FullQueue (QueueType Q)
{ int IndexTemp =  (Q.R == N -1) ? 0 : Q.R+1;
        return(Q.F == IndexTemp);
}

int EnQueue (QueueType &Q, ElementType x)
{       if (FullQueue(Q))
                { cout << "\nHàng đợi đầy !";
                  return 0;
                }
        if (Q.R == N-1) Q.R = 0; // xoay vòng chỉ số đuôi của hàng đợi
        else Q.R++;
        Gán (Q.mang[Q.R], x);
        // Cập nhật lại đầu hàng đợi rỗng sau khi thêm phần tử đầu tiên
        if (Q.F == -1) Q.F++;
        return 1;
}

int DeQueue (QueueType &Q, ElementType &x)
```

```
{       if (EmptyQueue(Q))
                { cout << "\nHàng đợi rỗng !"; return 0;
                }
        Gán (x, Q.mang[Q.F]);
        if (Q.F == Q.R)              // xóa trên hàng đợi chỉ còn một phần tử: Q sẽ rỗng !
                { Q.F = Q.R = -1;
                }
        else    if (Q.F == N-1)  Q.F = 0;          // xoay vòng chỉ số đầu hàng đợi
                else Q.F++;
        return 1;
}

int FrontQueue(QueueType  &Q, ElementType &x)
{       if (EmptyQueue(Q))
                { cout << "\nHàng đợi rỗng !"; return 0;
                }
        Gán (x, Q.mang[Q.F]);
        return 1;
}
```

c. Cài đặt hàng đợi bằng DSLK động

- ***Cài đặt cấu trúc dữ liệu***

Ta dùng kiểu dữ liệu con trỏ để cài đặt hàng đợi giống như cách cài đặt DSLK đơn.



- ***Các phép toán cơ bản***

Cách cài đặt các thao tác trên hàng đợi đều giống với các thao tác tương ứng trên DSLK đơn như: khởi tạo hàng đợi rỗng, kiểm tra xem hàng đợi có rỗng hay không, …

```
int EnQueue (LL  &Queue, ElementType x)
{       return InsertElementTailLL(Queue, x);
}

int DeQueue (LL &Queue, ElementType &x)
{       return RemoveHeadLL(Queue, x);
}

int FrontQueue(LL &Queue, ElementType &x)
{       if (EmptyQueue(Queue)) return 0;
        Gán(x, Queue.Head->Data);
        return 1;
```

}

d. <u>Ứng dụng của hàng đợi</u>

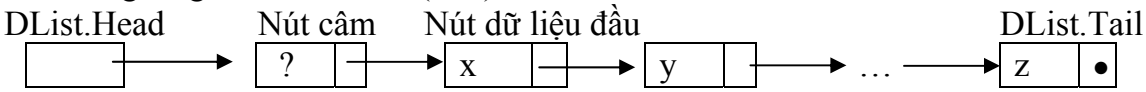
Hàng đợi có nhiều ứng dụng trong tin học như:

- Cơ chế vùng đệm cho các thao tác nhập – xuất trên bàn phím, máy in, thiết bị nhớ ngoài, …
- Hàng đợi lưu các tiến trình chờ được xử lý trong các hệ điều hành, trình biên dịch, …

III.4. Một số kiểu DSLK khác

III.4.1. <u>DSLK đơn có nút câm</u>

Qua các thao tác cơ bản trên DSLK đơn (không có nút câm trước đây), ta nhận thấy có sự *khác biệt* trong cách *xử lý giữa nút đầu* (không có nút đứng trước, ta thường qui ước *PredPtr* là NULL) *với các nút khác* (luôn có nút đứng trước *PredPtr*). Để *đơn giản khi viết* các thao tác trên (*khỏi phải phân biệt hai tình huống xử lý* đó) người ta tạo thêm một *nút giả (*hay *nút câm,* ta *không quan tâm đến dữ liệu* của nút này) đứng trước nút dữ liệu đầu tiên của DSLK đơn thông thường và gọi nó là DSLK (đơn) có nút câm.



Khi đó, các thao tác cơ bản trên DSLK có nút câm, sẽ được viết lại, trong một số trường hợp (chẳng hạn chèn, xóa) sẽ đơn giản hơn .

Cấp phát vùng nhớ cho một nút *(không quan tâm đến dữ liệu)*

```
NodePointer  CreateNode ()
{ NodePointer   new_ele;
        if ((new_ele = new NodeType) ==NULL)
                cout << "\nLỗi cấp phát vùng nhớ cho một nút mới !";
        else      new_ele ->Next = NULL;
        return new_ele;
}
```

- ***Khởi tạo một DSLK có nút câm rỗng***

```
LL CreateEmptyLL2 ()
{ LL List;
        List.Head = CreateNode();
        List.Tail = List.Head;
        return List;
}
```

- ***Kiểm tra một DSLK với nút câm có rỗng hay không***

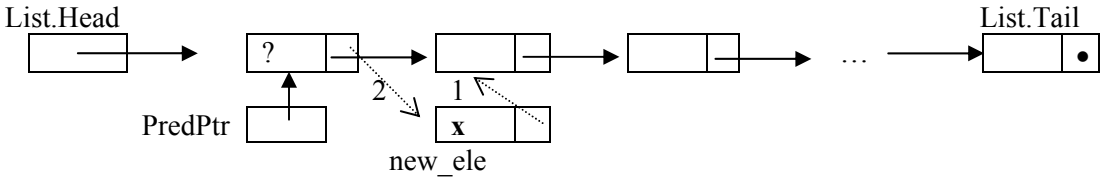
```
int EmptyLL2(LL  List)
{
        return(List.Head->Next == NULL);
}
```

- ***Duyệt qua một DSLK có nút câm***

```
int TraverseLL2(LL  List)
{ NodePointer CurrPtr = List.Head->Next;
        if (EmptyLL2(List)) return 0;
        else  { while (CurrPtr)
                  {   XửLý (CurrPtr);
                     CurrPtr = CurrPtr->Next;
                  }
                  return 1;
                }
}
```

- ***Thêm một phần tử x vào sau một nút được trỏ bởi con trỏ PredPtr***

** Thêm một nút mới vào sau một nút được trỏ bởi con trỏ PredPtr*




```
void InsertNodeAfterLL2(LL  &List, NodePointer  new_ele, NodePointer PredPtr)
{       new_ele->Next = PredPtr->next;
        PredPtr->Next = new_ele;
        if (PredPtr == List.Tail) List.Tail = new_ele;
        return  ;
}
```

** Thêm một phần tử x vào sau một nút được trỏ bởi con trỏ PredPtr*

```
int InsertElementAfterLL2(LL  &List, ElementType x, NodePointer PredPtr)
{ NodePointer  new_ele;
        if ((new_ele = CreateNodeLL(x)) == NULL) return 0;
        InsertNodeAfterLL2(List, new_ele, PredPtr);
        return 1;
}
```

Thêm một phần tử x vào đầu DSLK có nút câm

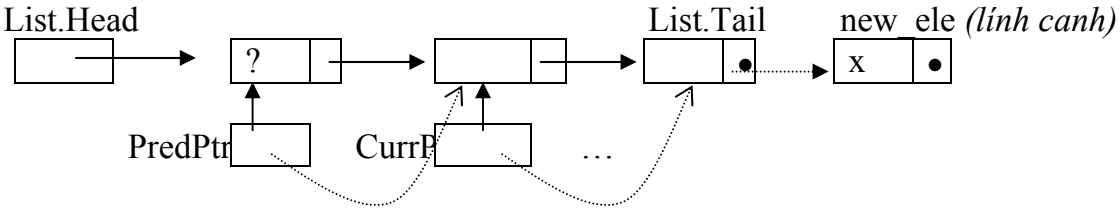
```
int InsertElementHeadLL2(LL  &List, ElementType x)
{       return InsertElementAfterLL2(List, x, List.Head);
}
```

Thêm một phần tử x vào cuối DSLK có nút câm

```
int InsertElementTailLL2(LL  &List, ElementType x)
{       return InsertElementAfterLL2(List, x, List.Tail);
}
```

- ***Tìm kiếm một phần tử trên DSLK đơn có nút câm***

Tìm một phần tử *x* trong DSLK *List*. Nếu *tìm thấy* thì, thông qua đối cuối của hàm, *trả về địa chỉ PredPtr của nút đứng trước nút tìm thấy đầu tiên*. Để *tăng tốc độ tìm kiếm* (bằng cách giảm số lần so sánh trong biểu thức điều kiện của vòng lặp), ta đặt *thêm lính canh* ở cuối List.



- ***Thuật toán tìm kiếm tuyến tính (có lính canh) trên dãy chưa được sắp:***

Boolean ***SearchLinearLL2****(List, x, &PredPtr)*

. Chèn nút mới new_ele chứa x vào cuối List (đóng vai trò lính canh)
. PredPtr = List.Head;
  CurrPtr = List.Head->Next; // PredPtr đứng kề trước CurrPtr
. Trong khi (CurrPtr->Data ≠ x) thực hiện
  { PredPtr = CurrPtr; CurrPtr = CurrPtr->Next;
  }
. if (CurrPtr ≠ new_ele) Thấy = True; // Thông báo thấy x;
  else Thấy = False; // Thông báo không thấy x;
. Xóa nút (new_ele) đứng sau nút được trỏ bởi List.Tail;
. Trả về trị Thấy;

- ***Cài đặt***

```
int SearchLinearLL2(LL List, ElementType x, NodePointer &PredPtr)
{ NodePointer  CurrPtr = List.Head->Next,  OldTail = List.Tail,
               new_ele = InsertElementTailLL2(List, x);
      PredPtr = List.Head;
  int Thấy;
      while (SoSánh(CurrPtr->Data, x) != 0)
        { PredPtr = CurrPtr ;  CurrPtr = CurrPtr->Next;
        }
       if (CurrPtr != new_ele) Thấy = 1;          // thấy thật sự
```

```
        else Thấy = 0;                          // thấy giả hay không thấy !
        RemoveAfterLL2(List, OldTail, x);       // xóa nút new_ele;
        return Thấy;
}
```

- ***Xóa một nút sau một nút được trỏ bởi con trỏ PredPtr***

```
int RemoveAfterLL2(LL  &List, NodePointer PredPtr, ElementType &x)
{ NodePointer Temp;
        if (EmptyLL2(List))
                { cout << "\nDS rỗng !";  return 0;
                }
        Temp = PredPtr->Next;
        if (Temp == NULL) return 0;        // không xóa được nút sau nút cuối ?!
        else PredPtr->Next = Temp->Next;
        if (Temp == List.Tail) List.Tail = PredPtr;        //nếu xóa đuôi, cần cập
        nhật lại đuôi
        Gán(x, Temp->Data);
        delete Temp;
        return 1; // xóa thành công
}
```

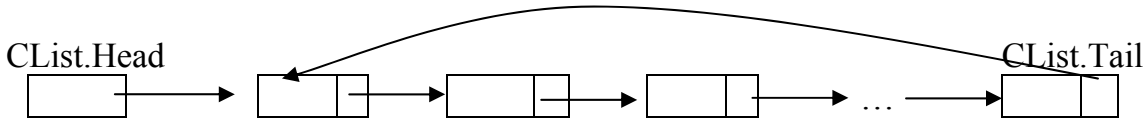
Việc viết lại các thao tác cơ bản còn lại trên DSLK đơn có nút câm được xem như bài tập. Qua đó, ta thấy rõ mối liên quan mật thiết giữa cấu trúc dữ liệu và thuật toán, được thể hiện qua "công thức" của *Niklaus Wirth*:

Cấu trúc dữ liệu + Thuật toán = Chương trình

III.4.2. DSLK vòng

DSLK vòng là DSLK mà nút cuối là nút kề trước của nút đầu.

Nếu cài đặt DSLK vòng bằng kiểu con trỏ thì con trỏ của nút cuối trỏ đến nút đầu tiên. Trong DSLK vòng, ta có thể lấy bất cứ nút nào làm nút đầu tiên xuất phát. Cấu trúc dữ liệu cho mỗi nút của DSLK vòng hoàn toàn giống như DSLK đơn.



Một số thao tác cơ bản cho DSLK vòng sẽ được viết lại sau đây, các thao tác khác được xem như bài tập.

- ***Khởi tạo một DSLK vòng rỗng***

```
LL CreateEmptyCLL ()
{ LL CList;
        CList.Head = CList.Tail = NULL;
        return List;
}
```

- ***Kiểm tra một DSLK vòng có rỗng hay không***

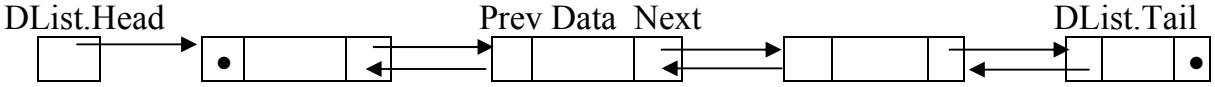
```
int EmptyCLL(LL  CList)
{
        return(CList.Head == NULL  && CList.Tail == NULL);
}
```

- ***Duyệt qua một DSLK vòng***

```
int TraverseCLL(LL  CList)
{ NodePointer CurrPtr = CList.Head
        if (EmptyCLL(CList)) return 0;
        do
       { XửLý (CurrPtr);
         CurrPtr = CurrPtr->Next;
        } while (CurrPtr->Next != Clist.Head);
        return 1;
}
```

III.4.3. DSLK đối xứng

Trong nhiều thao tác trên kiểu DSLK đơn, khi làm việc với một nút ta *cần biết nút đứng kề trước* của nó. Lý do là DSLK đơn chỉ có một liên kết đi theo một chiều từ nút đứng trước đến nút đứng sau. Để *tăng độ linh hoạt* trong các thao tác trên DSLK, có thể di chuyển từ đầu đến đuôi của danh sách hay ngược lại, ta xét kiểu *DSLK đối xứng* (hay DSLK kép) mà *mỗi nút có hai trường liên kết ngược chiều nhau*, một liên kết chỉ đến nút đứng sau và liên kết kia chỉ đến nút đứng trước.



a. Cấu trúc dữ liệu biểu diễn DSLK đối xứng

Trong C hay C++, mỗi nút của DSLK đối xứng được cài đặt bởi cấu trúc sau:

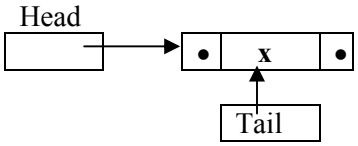
```
typedef    ....   ElementType;          // Kiểu dữ liệu cơ sở của mỗi phần tử
typedef struct Dnode {ElementType  Data;
```

```
                    struct Dnode *Next, *Prev;
                  } DNodeType;
typedef  DNodeType *DNodePointer;
typedef  struct { DNodePointer Head, Tail;
                } DLL;
DLL  DList;
```

b. Các thao tác cơ bản trên DSLK đối xứng

Các thao tác cơ bản về sau sẽ sử dụng thủ tục cấp phát động vùng nhớ cho một nút của DSLK đối xứng sau đây:

- ***Cấp phát vùng nhớ chứa dữ liệu x cho một nút của DSLK đối xứng***



- ***Thuật toán***

DNodePointer ***CreateNodeDLL*** *(x)*

. Cấp phát vùng nhớ cho một nút new_ele;

. new_ele ->Data = x; new_ele ->Next = NULL; new_ele ->Prev = NULL;

. Trả về new_ele;

- ***Cài đặt***

```
DNodePointer  CreateNodeDLL (ElementType x)
{ DNodePointer   new_ele;
        if ((new_ele = new DNodeType) ==NULL)
                cout << "\nLỗi cấp phát vùng nhớ cho một nút mới !";
        else      { Gán(new_ele ->Data, x);
                    new_ele ->Next = new_ele ->Prev = NULL;
                  }
        return new_ele;
}
```

- ***Khởi tạo một DSLK đối xứng rỗng.***
- ***Thuật toán***

DLL ***CreateEmptyDLL*** *()*

. DList.Head = DList.Tail = NULL;

. Trả về DList;

- ***Cài đặt***

```
DLL CreateEmptyDLL ()
{ DLL List;
        DList.Head = DList.Tail = NULL;
        return DList;
```

```
}
```

- ***Kiểm tra một DSLK đối xứng có rỗng hay không***

- ***Thuật toán***

Boolean ***EmptyDLL****(DLL DList)*
if (DList.Head == NULL)
// hay (DList.Head == NULL) && (DList.Tail == NULL). Tại sao ? Hãy so sánh !
Trả trị True; // DList rỗng;
else Trả trị False; // DList khác rỗng;

- ***Cài đặt***

```
int EmptyDLL(DLL DList)
{       return(DList.Head == NULL);
  // hay return ((DList.Head == NULL) && (DList.Tail == NULL));
}
```

- ***Duyệt qua một DSLK đối xứng***

Ta có thể duyệt Dlist theo chiều thuận (hay ngược) tùy theo chiều con trỏ Next (hay Prev).

- ***Thuật toán***

TraverseLL*(DList)*
. CurrPtr = DList.Head; // hay CurrPtr = DList.Tail;
. Trong khi chưa hết DSLK thực hiện:
{ XửLý nút được trỏ bởi CurrPtr;
CurrPtr = CurrPtr->Next; // chuyển đến nút kề sau
// hay CurrPtr = CurrPtr->Prev; chuyển đến nút kề trước
}

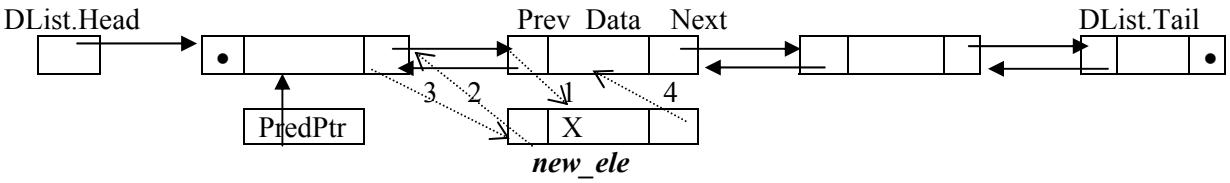
- ***Cài đặt***

```
int TraverseDLL(DLL DList)
{ DNodePointer CurrPtr = DList.Head; // hay CurrPtr = DList.Tail;
        if (EmptyDLL(DList)) return 0;
        else { while (CurrPtr != NULL)  // hoặc   while (CurrPtr)
                 {   XửLý (CurrPtr);
                    CurrPtr = CurrPtr->Next; // hay CurrPtr = CurrPtr->Prev;
                 }
                 return 1;
               }
}
```

```
void  XửLý(DNodePointer CurrPtr)
{       // Xử  lý nút CurrPtr  tùy theo từng yêu cầu cụ thể
        return ;
}
```

- ***Thêm một phần tử mới vào DSLK đối xứng***

** Thêm một phần tử vào sau một nút được trỏ bởi con trỏ PredPtr*
(nếu PredPtr == NULL thì chèn phần tử vào đầu DSLK)



- ***Thuật toán:*** *Thêm một nút new_ele vào sau một nút được trỏ bởi PredPtr*

InsertNodeAfterDLL(&*DList, new_ele, PredPtr)*

```
. if (PredPtr)
            { new_ele->Next = PredPtr->Next; new_ele->Prev = PredPtr;
              PredPtr->Next = new_ele;
              if (new_ele->Next)  (new_ele->Next)->Prev = new_ele;
// else: trường hợp chèn new_ele vào đuôi DList, không cập nhật nút sau nút new_ele
            }
  else      // chèn new_ele vào đầu Dlist
            { new_ele->Next = DList.Head;
              if (DList.Head) DList.Head->Prev = new_ele;
            // else DS rỗng !
              DList.Head = new_ele; //cập nhật lại nút đầu DS
            }
 // nếu chèn nút mới vào đuôi, cần cập nhật lại đuôi mới
. if (PredPtr == DList.Tail) DList.Tail = new_ele;
```

- ***Cài đặt***

```
void InsertNodeAfterDLL(DLL &DList, DNodePointer  new_ele,DNodePointer PredPtr)
{       if (PredPtr)
            { new_ele->Next = PredPtr->next; new_ele->Prev = PredPtr;
              PredPtr->Next = new_ele;
              if (new_ele->Next)  (new_ele->Next)->Prev = new_ele;
            }
        else    { new_ele->Next = DList.Head;
              if (DList.Head) DList.Head->Prev = new_ele;
              DList.Head = new_ele;
            }
        if (PredPtr == DList.Tail) DList.Tail = new_ele;
        return  ;
}
```

- ***Thuật toán:*** *Thêm phần tử x vào sau một nút được trỏ bởi con trỏ PredPtr*

DNodePointer ***InsertElementAfterDLL*** *(&DList, x, PredPtr)*

. new_ele = CreateNodeDLL (x);

. if (new_ele ≠ NULL) Thêm nút new_ele vào sau nút được trỏ bởi PredPtr;

. Trả về trị new_ele;

- ***Cài đặt***

```
DNodePointer InsertElementAfterDLL(DLL &DList,ElementType x,DNodePointer PredPtr)
{ DNodePointer new_ele;
    if ((new_ele = CreateNodeDLL (x)))
        InsertNodeAfterDLL (DList, new_ele, PredPtr);
    return (new_ele);
}
```

Tương tự, ta có thao tác thêm một nút (hay phần tử) vào trước một nút được trỏ bởi con trỏ *SuccPtr (bài tập)*.

- <u>*Thêm một phần tử vào cuối một DSLK đối xứng*</u>
- ***Thuật toán:*** *Thêm một nút new_ele vào cuối DSLK DList*
  InsertNodeTailDLL(&*DList, new_ele)*

. Thêm nút new_ele vào sau nút được trỏ bởi DList.Tail.

- ***Cài đặt***

```
void InsertNodeTailDLL(DLL &DList, DNodePointer new_ele)
{
    InsertNodeAfterDLL (DList, new_ele, DList.Tail);
    return ;
}
```

- ***Thuật toán:*** *Thêm phần tử x vào cuối Dlist*
  DNodePointer ***InsertElementTailDLL*** *(&DList, x)*

. Thêm phần tử x vào sau nút được trỏ bởi DList.Tail.

- ***Cài đặt***

```
DNodePointer InsertElementTailDLL (DLL &DList, ElementType x)
{
    return (InsertElementAfterDLL (DList, x, DList.Tail));
}
```

- <u>*Thêm một phần tử vào đầu một DSLK đối xứng*</u>
- ***Thuật toán:*** *Thêm một nút new_ele vào đầu DSLK DList*
  InsertNodeHeadDLL(&*DList, new_ele)*

. Thêm nút new_ele vào đầu DList (hay sau nút được trỏ bởi NULL).

- ***Cài đặt***

```
void InsertNodeHeadDLL(DLL  &DList, DNodePointer new_ele)
{
        InsertNodeAfterDLL (DList, new_ele, NULL);
        return ;
}
```

- ***Thuật toán:*** *Thêm phần tử x vào đầu Dlist*
  DNodePointer ***InsertElementHeadDLL*** *(&DList, x)*
  . Thêm phần tử x vào đầu DList (hay sau nút được trỏ bởi NULL).

- ***Cài đặt***

```
DNodePointer InsertElementHeadDLL (DLL  &DList, ElementType x)
{
        return (InsertElementAfterDLL (DList, x, NULL));
}
```

- ***Tìm kiếm một phần tử trên DSLK đối xứng***

Thuật toán tìm kiếm trên DSLK đối xứng *hoàn toàn tương tự* như trên DSLK đơn. Nếu *tìm thấy* phần tử trên danh sách thì *trả về con trỏ chứa địa chỉ nút vừa thấy*, *nếu không thấy* trả về *NULL.*

- ***Thuật toán tìm kiếm tuyến tính (có lính canh) trên dãy chưa được sắp:***

DNodePointer ***SearchLinearDLL****(DList, x)*
. Chèn nút mới new_ele chứa x vào cuối DList (đóng vai trò lính canh)
. CurrPtr = DList.Head;
. Trong khi (CurrPtr->Data ≠ x) thực hiện
CurrPtr = CurrPtr->Next;
. if (CurrPtr ≠ new_ele) Thông báo thấy x;
else { Thông báo không thấy x; // thấy giả !
CurrPtr = NULL;
}
. Xoá nút new_ele; Trả về CurrPtr;

- ***Cài đặt***

```
DNodePointer SearchLinearDLL(DLL  DList, ElementType x)
{ DNodePointer  new_ele = InsertElementTailDLL(DList, x),
        CurrPtr = DList.Head;
        while (SoSánh(CurrPtr->Data, x) != 0)
                CurrPtr = CurrPtr->Next;
        if (CurrPtr == new_ele) CurrPtr = NULL;          // không thấy
        RemoveNodeDLL(DList, new_ele);
        return CurrPtr;
```

}

- ***Thuật toán tìm kiếm tuyến tính (có lính canh) trên dãy được sắp** (tăng)**:***

DNodePointer ***SearchLinearOrderDLL****(DList, x)*
. Chèn nút mới new_ele chứa x vào cuối DList (đóng vai trò lính canh)
. CurrPtr = DList.Head;
. Trong khi (CurrPtr->Data < x) thực hiện
CurrPtr = CurrPtr->Next;
. if ((CurrPtr ≠ new_ele) and (CurrPtr->Data ≡ x)) Thông báo thấy x;
else { Thông báo không thấy x;
CurrPtr = NULL;
}
. Xoá nút new_ele; Trả về CurrPtr;

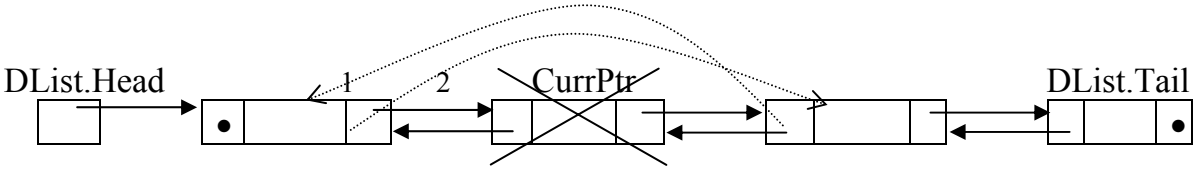
- ***Cài đặt***

```
DNodePointer SearchLinearOrderDLL(DLL List, ElementType x)
{ DNodePointer  new_ele = InsertElementTailDLL(DList, x),
        CurrPtr = DList.Head;
        while (SoSánh(CurrPtr->Data, x) < 0)
              CurrPtr = CurrPtr->Next;
         if ((CurrPtr == new_ele) || (SoSánh(CurrPtr->Data, x) > 0))) CurrPtr = NULL;
        RemoveNodeDLL(DList, new_ele);
        return CurrPtr;
}
```

Với DSLK đối xứng, ta có thể *tìm kiếm theo chiều ngược lại*, bằng cách xuất phát từ DList.Tail và tìm từ cuối về đầu theo trường con trỏ *Prev (bài tập)*.

- ***Xóa một phần tử khỏi DSLK đối xứng***

** Xóa một nút được trỏ bởi con trỏ CurrPtr*



- ***Thuật toán***

int ***RemoveNodeDLL****(&DList, CurrPtr, &**x**)*
. if (CurrPtr == DList.Head) // xóa nút đầu
// hay dở hơn, tại sao ? if (CurrPtr->Prev == NULL)
{ DList.Head = CurrPtr->Next;
if (DList.Head ==NULL) // xóa trên DS chỉ có 1 nút

```
                    DList.Tail =  NULL;
                else  DList.Head->Prev = NULL;
            }
        else  { if (CurrPtr == DList.Tail)          //xóa nút cuối
                 // hay dở hơn, tại sao ?  if (CurrPtr->Next == NULL)
                    { DList.Tail =  CurrPtr->Prev;
      //không cần ?   if (DList.Tail ==NULL) DList.Head = NULL;//xóa DS có 1 nút
                      DList.Tail->Next = NULL;
                    }
                else  {(CurrPtr->Next)->Prev =  CurrPtr->Prev;
                       (CurrPtr->Prev)->Next =  CurrPtr->Next;
                    }
            }
      . Gán(x, Temp->Data);  delete CurrPtr;
```

- ***Cài đặt***

```
int RemoveNodeDLL(DLL  &DList, DNodePointer CurrPtr, ElementType &x)
{
      if (EmptyDLL(DList))
            { cout << "\nDS rỗng !";   return 0;
            }
       if (CurrPtr->Prev == NULL)              //xóa nút đầu
            { DList.Head =  CurrPtr->Next;
              if (DList.Head ==NULL)                  // xóa trên DS chỉ có 1 nút
                    DList.Tail =  NULL;
              Else DList.Head->Prev = NULL;
            }
        else  { if (CurrPtr->Next == NULL)     //xóa nút cuối
                    { DList.Tail =  CurrPtr->Prev;
                      DList.Tail->Next = NULL;
                    }
                else  {(CurrPtr->Next)->Prev =  CurrPtr->Prev;
                       (CurrPtr->Prev)->Next =  CurrPtr->Next;
                    }
            }
      Gán(x, Temp->Data);
      delete CurrPtr;
      return 1; // xóa thành công
}
```

* *Xóa nút đầu của DSLK đối xứng*

```
int RemoveHeadDLL(DLL &DList, ElementType &x)
{       return RemoveNodeDLL (DList, DList.Head, x);
}
```

* *Xóa nút cuối của DSLK đối xứng*

```
int RemoveTailDLL(DLL &DList, ElementType &x)
{       return RemoveNodeDLL (DList, DList.Tail, x);
}
```

* *Xóa một phần tử x khỏi DSLK*

- ***Thuật toán:***

int ***RemoveElementDLL(&****DList, x)*

. Tìm *x* trong D*List*, nếu thấy thì trả về biến con trỏ *CurrPtr* chỉ đến nút tìm thấy.

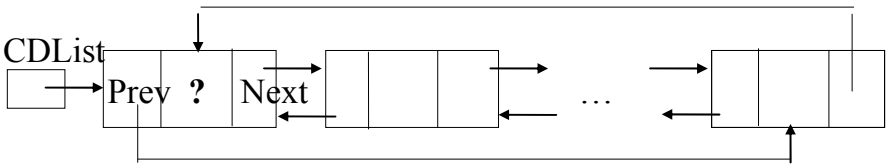
. Xóa nút được trỏ bởi *CurrPtr*.

- ***Cài đặt***

```
int RemoveElementDLL(DLL &DList, ElementType x)
{ DNodePointer CurrPtr;
        if ((CurrPtr = SearchLinearDLL(DList, x) == NULL) return 0; // không thấy
        else return RemoveNodeDLL (DList, CurrPtr, x);
}
```

Việc *hủy nút cuối* trên *DSLK đối xứng có chi phí O(1)*, chứ không phải tốn chi phí O(n) như đối với DSLK đơn. Tuy vậy, việc cài đặt một dãy các đối tượng bằng DSLK đối xứng *tốn bộ nhớ lớn gấp đôi để lưu trữ hai liên kết* và việc *cập nhật* cũng *nặng nề hơn*.

Nếu kết hợp các tính chất: thêm nút câm, vòng và đối xứng thì ta sẽ được kiểu DSLK "*vòng đôi*". Hãy viết các thao tác cơ bản trên kiểu danh sách này (*bài tập*).



c. Ứng dụng của DSLK đối xứng

Ta có thể *dùng DSLK đối xứng* để cài đặt *hàng đợi hai đầu* (Dequeue – Double ended queue). Tất nhiên, ta cũng có thể biểu diễn Dequeue bằng DSLK đơn nhưng bất tiện hơn. Hàng đợi hai đầu sẽ được sử dụng trong các thuật toán tìm kiếm trong lý thuyết đồ thị và trí tuệ nhân tạo.

Hàng đợi hai đầu là danh sách mà việc *thêm và hủy đều có thể thực hiện ở hai đầu* danh sách, trên đó có các thao tác chính sau:

- Thêm phần tử *x* vào đầu hàng đợi hai đầu Dequeue:
InsertHead (Dequeue, x) hay chính là *InsertElementHeadDLL*(Dequeue, x);
- Thêm phần tử *x* vào cuối Dequeue:
InsertTail (Dequeue, x) hay chính là *InsertElementTailDLL*(Dequeue, x);
- Lấy ra phần tử ở đầu Dequeue:
RemoveHead (Dequeue, x) hay chính là *RemoveHeadDLL*(Dequeue, x);
- Lấy ra phần tử ở cuối Dequeue:
RemoveTail (Dequeue, x) hay chính là *RemoveTailDLL*(Dequeue, x);

Ngoài ra, trên Dequeue còn hỗ trợ các thao tác sau:

- Kiểm tra xem Dequeue có rỗng không:
EmptyDequeue(Dequeue) hay chính là *EmptyDLL*(Dequeue);
- Xem giá trị ở đầu Dequeue mà không hủy nó khỏi Dequeue:
Head(Dequeue) hay chính là *Dequeue.Head;*
- Xem giá trị ở cuối Dequeue mà không hủy nó khỏi Dequeue:
Tail(Dequeue) hay chính là *Dequeue.Tail;*

Ta có thể *dùng hàng đợi hai đầu* để *biểu diễn ngăn xếp* và *hàng đợi* như được minh họa trong bảng sau. Lưu ý rằng *tất cả các thao tác này trên Dequeue đều có độ phức tạp hằng* O(1).

Dequeue	Queue	Stack
InsertHead (Dequeue, x)		***Push*** (Stack, x)
InsertTail (Dequeue, x)	***EnQueue*** (Queue, x)	
RemoveHead (Dequeue, x)	***DeQueue*** (Queue, x)	***Pop*** (Stack, x)
RemoveTail (Dequeue, x)		
EmptyDequeue(Dequeue)	***EmptyQueue*** (Queue)	***EmptyStack*** (Stack)
Head(Dequeue, x)	***Front*** (Queue)	***Top*** (Stack, x)
Tail(Dequeue, x)		

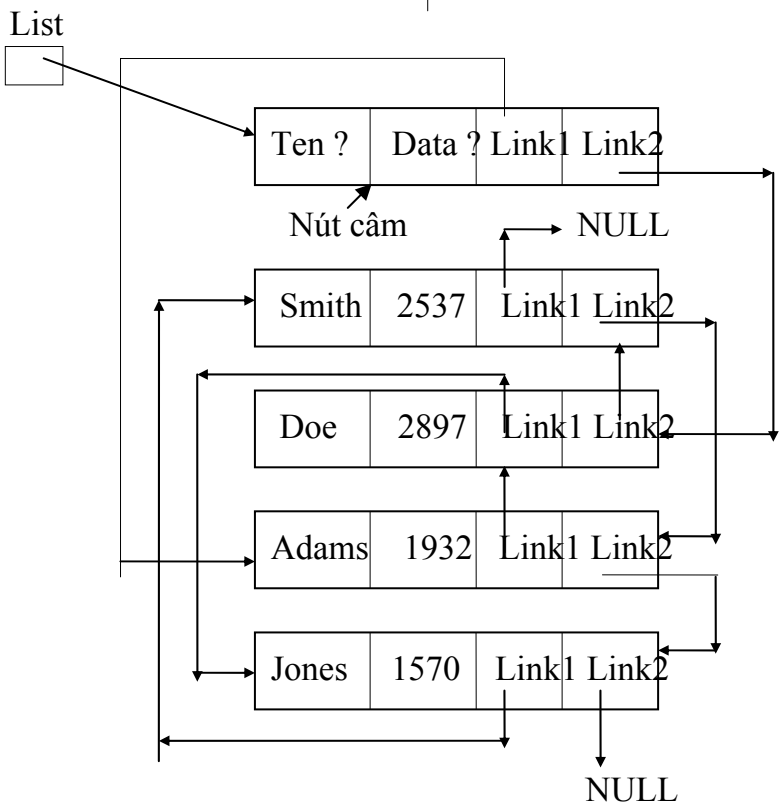
III.4.4. Danh sách đa liên kết

Danh sách đa liên kết là danh sách mà *mỗi nút* của nó, *ngoài* thành phần *dữ liệu* (có thể có nhiều trường), còn gồm *nhiều thành phần liên kết khác phục vụ cho những mục đích khác nhau.*

Chẳng hạn, ta có thể dùng *danh sách liên kết động có hai liên kết* (*không nhất thiết phải đối xứng*) để *lưu trữ* và *sắp xếp dãy các mẫu tin theo hai quan hệ thứ tự khác nhau, chẳng hạn theo hai trường khóa* khác nhau nào đó.

Ví dụ: Ta muốn lưu danh sách sau, sao cho theo những trường khóa khác nhau chúng được sắp xếp theo những thứ tự nào đó.

Tên	Mã Số
Smith	2537
Doe	2897
Adams	1932
Jones	1570



Với mỗi mẫu tin, ngoài trường dữ liệu, ta còn lưu thêm hai trường con trỏ: *Link1* hay *NextTên* để *sắp tăng* các mẫu tin này theo trường *Tên*, còn *Link2* hay *NextMãSố* để *sắp giảm* các mẫu tin này theo trường MãSố.

Ta dùng danh sách đa (trong ví dụ này là hai) liên kết có nút câm để lưu trữ danh sách các mục dữ liệu. Nếu đi *theo Link1*, ta được danh sách *tăng theo thứ tự Tên*; nếu đi *theo Link2*, ta được danh sách *giảm theo thứ tự Mã Số*.

a. Cài đặt cấu trúc dữ liệu cho DS đa liên kết

```
typedef unsigned long So;
typedef struct {char Ten[MAX_TEN];
```

```
                    So     MaSo;
                  } ElementType;
        typedef  struct MultiNode  *MultiPtr;
        struct MultiNode   { ElementType Data;
                             MultiPtr  NextTen, NextMaSo;
                           };
        MultiPtr  MList;
```

b. Vài thao tác cơ bản trên DS đa liên kết

Cấp phát vùng nhớ cho một nút của DS đa liên kết

```
MultiPtr CreateNodeML()
{ MultiPtr new_ele;
        if ((new_ele = new MultiNode) == NULL)
                cout << "\n Lỗi cấp phát bộ nhớ cho một nút của DS đa LK !";
        else      new_ele->NextTen = new_ele->NextMaSo = NULL;
        return Temp;
}
```

- ***Thủ tục thêm một nút vào DS đa liên kết***

Sau khi thêm mẫu tin mới {Ten0, MaSo0} vào DSLK cũ, vẫn bảo đảm thứ tự tăng theo Tên và giảm theo MãSố trong DSLK mới thu được.

```
int  InsertOrderMulti(MultiPtr MList, char Ten0[MAX_TEN], So MaSo0)
{ MultiPtr  new_ele, PredPtr, CurrPtr;
        if  ((new_ele = CreateNodeML()) == NULL) return 0;
        Gan ((new_ele->Data).Ten,Ten0); Gan((new_ele ->Data).MaSo,MaSo0);
// Tìm vị trí chèn (tăng) new_ele theo trường NextTen
        PredPtr = MList;
        CurrPtr = PredPtr->NextTen;
        while (CurrPtr  && SoSanh((CurrPtr ->Data).Ten, Ten0) < 0)
           {    PredPtr = CurrPtr;
                CurrPtr = CurrPtr ->NextTen;
           }
         PredPtr->NextTen  = new_ele;
         new_ele->NextTen  = CurrPtr;
// Tìm vị trí chèn (giảm) new_ele theo trường NextMaSo
        PredPtr = MList;
        CurrPtr  = PredPtr->NextMaSo;
        while (CurrPtr  && (CurrPtr ->Data).MaSo > MaSo0)
           {    PredPtr =  CurrPtr;
                CurrPtr =  CurrPtr ->NextMaSo;
           }
         PredPtr->NextMaSo = new_ele;
```

```
        new_ele ->NextMaSo = CurrPtr  ;
        return 1;
}
```

- ***Thủ tục xóa một nút từ DS đa liên kết***

```
int  DeleteOrderMulti(MultiPtr Mlist, char Ten0[MAX_TEN], So MaSo0)
{ MultiPtr  LưuVịTrí, PredPtr, CurrPtr;
        // Tìm vị trí trùng tên Ten0 theo trường NextTen
        PredPtr = MList;
        CurrPtr = MList->NextTen;
        while (CurrPtr  && SoSanh((CurrPtr ->Data).Ten, Ten0) != 0
                        && (CurrPtr->Data).MaSo != MaSo0)
                if  (SoSanh(CurrPtr ->Data).Ten, Ten0) > 0)      // không thấy
                        CurrPtr = NULL;
                else    { PredPtr = CurrPtr;                              // chưa thấy
                          CurrPtr = CurrPtr ->NextTen;
                        }
        if  (CurrPtr == NULL) return 0;    // Không thấy nên không xóa được
        LưuVịTrí = CurrPtr;
        PredPtr->NextTen = CurrPtr ->NextTen; // Đã thấy tên trùng với Ten0

        // Tìm vị trí trùng mã số MaSo0  theo trường NextMaSo
        PredPtr = Mlist;
        CurrPtr = MList->NextMaSo;
        while (CurrPtr != LưuVịTrí)
             { PredPtr = CurrPtr;
               CurrPtr = CurrPtr ->NextMaSo;
             }
        PredPtr->NextMaSo = CurrPtr ->NextMaSo;
        delete  LưuVịTrí;
        return 1;
}
```

III.4.5. Một số ứng dụng khác của DSLK

a. DS có thứ tự và DS tổ chức lại

Danh sách có thứ tự (Order List) là loại danh sách mà các phần tử của nó được tổ chức lưu trữ *thỏa mãn một quan hệ thứ tự* nào đó dựa trên các thành phần dữ liệu của chúng *nhằm* phục vụ cho việc *khai thác dữ liệu* (chẳng hạn *tìm kiếm và cập nhật*) *được nhanh chóng và thuận lợi hơn*. Với kiểu DS này, hầu hết các thao tác cơ bản trên DSLK đều được giữ nguyên. Riêng thao tác *chèn (*và *xoá)*

một phần tử mới x vào một DSLK OList đã được sắp cho trước cần được viết lại *để thu được danh sách mới vẫn được sắp*.

- Trước hết, ta xây dựng thuật toán *tìm một nút PredPtr xa nhất chứa dữ liệu* trên DSLK có thứ tự (giả sử tăng) O*List sao cho PredPtr->Data < x, nếu không* có nút thỏa mãn tính chất này (trường hợp x ≤ dữ liệu nút đầu tiên) ta qui ước cho *PredPtr = NULL*. Sau đó, ta sẽ chèn *x* vào sau nút *PredPtr*.

- ***Thuật toán:***

NodePointer ***SearchLinearOrderLL****(OList, x)*
. Chèn nút mới new_ele chứa x vào cuối OList (đóng vai trò lính canh)
. PredPtr = NULL; CurrPtr = OList.Head;
. Trong khi (CurrPtr->Data < x) thực hiện
{ PredPtr = CurrPtr ; CurrPtr = CurrPtr->Next;
}
. Xóa nút new_ele (sau nút OList.Tail); Trả về PredPtr;

- ***Cài đặt***

```
NodePointer SearchLinearOrderLL(LL  OList, ElementType x)
{ NodePointer  CurrPtr = OList.Head, PredPtr = NULL,
        new_ele = InsertElementTailLL(OList, x);
        while (SoSánh(CurrPtr->Data, x) < 0)
          { PredPtr = CurrPtr;
            CurrPtr = CurrPtr->Next;
          }
        RemoveAfterLL(OList, OList.Tail, x);
        return PredPtr;
}
```

- ***Chèn tăng một phần tử x vào DSLK đơn OList đã sắp tăng***

- ***Thuật toán***

InsertOrderLL ***(&****OList, x)*
. PredPtr = *SearchLinearOrderLL*(OList, x);
. Thêm phần tử *x* vào sau nút được trỏ bởi PredPtr;

- ***Cài đặt***

```
int InsertOrderLL (LL &OList, ElementType x)
{ NodePointer  PredPtr = SearchLinearOrderLL(OList, x);
        return InsertElementAfterLL (OList, x, PredPtr);
}
```

Danh sách có thứ tự có thể được *cài đặt bằng DSLK đối xứng*, khi đó cần viết lại thao tác chèn tương ứng với cách cài đặt này (*bài tập*). Khi đó ta có thể *dùng DSLK đối xứng* và *có thứ tự để cài đặt hàng đợi có ưu tiên* (được ứng dụng nhiều trong tin học, chẳng hạn việc quản lý những tiến trình trong các hệ điều hành).

Trên thực tế, trong nhiều trường hợp khi khai thác dữ liệu trên một DSLK đã có một quan hệ thứ tự cho trước trên miền dữ liệu chung của các phần tử, ta thấy hiện tượng sau thường xảy ra: có *nhiều phần tử* (có thể *không ở gần đầu* danh sách) *được khai thác thường xuyên hơn* các phần tử khác. Khi đó, *để giảm chi phí tìm kiếm* trong khai thác dữ liệu, ta có thể *tổ chức lại danh sách* (gọi là *danh sách tổ chức lại*) bằng cách *chèn những phần tử này vào đoạn đầu* của danh sách. Nhưng với cách tổ chức như thế, *quan hệ thứ tự cũ bị phá vỡ*, do đó ta *không tận dụng được các thao tác hiệu quả trên DSLK được sắp thứ tự*, dẫn đến *chi phí tìm kiếm các phần tử khác tăng lên*!

Một cách tiếp cận khác là *tổ chức lại dữ liệu, bằng cách* ***tạo ra quan hệ thứ tự mới*** *dựa trên việc bổ sung thêm một thành dữ liệu cho mỗi nút là số lần mà nó được khai thác với độ ưu tiên nào đó* cho thỏa đáng *so với độ ưu tiên của các thành phần dữ liệu khác*. Tất nhiên, cách tổ chức này tuy làm *tăng tốc độ tìm kiếm* khi khai thác dữ liệu, nhưng lại phải *trả giá về chi phí bộ nhớ tăng lên* ! May mắn cho chúng ta là không gian nhớ giành thêm để lưu trữ số lần khai thác mỗi mục dữ liệu thường không đáng kể so với kích thước rất lớn của dữ liệu trong các bài toán thực tế thường gặp khi lưu trữ các cơ sở dữ liệu lớn. Đó là vấn đề thường xuyên xảy ra khi cải tiến thuật toán: *việc giảm chi phí về thời gian thường tăng chi phí về không gian bộ nhớ và ngược lại*! Chọn cách tổ chức kiểu dữ liệu nào sẽ tùy thuộc vào đặc điểm của từng bài toán và mục đích tiết kiệm tài nguyên về khía cạnh cụ thể nào là quan trọng nhất.

b. Biểu diễn tập hợp bằng DSLK (có nút câm)

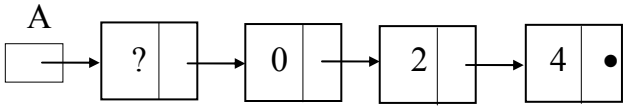
Như đã biết, ta có thể biểu diễn tập hợp theo *dãy bit* bằng cách dùng một mảng các bit để biểu diễn tập hợp con bất kỳ của một tập phổ dụng. Hạn chế của cách biểu diễn nàylà khi tập hợp con thực sự rất bé nhưng tập phổ dụng lại rất lớn sẽ gây lãng phí bộ nhớ.

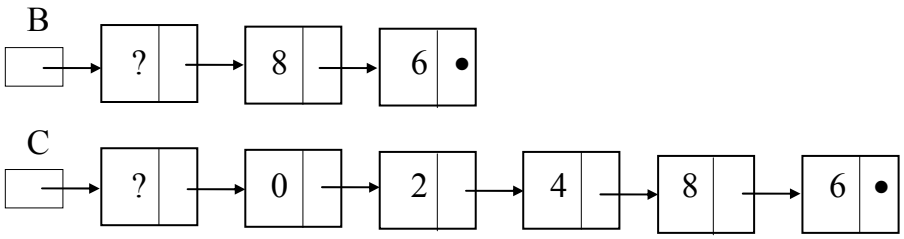
Sau đây, ta đưa ra một cách tiếp cận khác: dùng *DSLK đơn có nút câm* để biểu diễn tập hợp, trong đó ta không phân biệt thứ tự của các phần tử cũng như không có sự trùng lặp giữa các phần tử trong DSLK.

<u>*Dùng DSLK với nút câm cài đặt tập hợp*</u>

<u>*Ví dụ*</u>: Ta biểu diễn các tập hợp sau bằng DSLK đơn có nút câm:

A = {0,2,4}, B = {8,6}, C = A U B.





- ***Thủ tục thêm một phần tử vào tập hợp***

```
int AddElement(LL  S, ElementType x)
{
        return InsertElementHeadLL2(S, x);
}
```

- ***Kiểm tra (hay tìm kiếm) xem một phần tử x có thuộc tập S hay không***

```
int IsAMember (LL  S, ElementType x)
{ NodePointer PredPtr;
        return SearchLinearLL2(S, x, PredPtr);
}
```

- ***Phép hợp A U B***

```
int  Union(LL  A, LL  B, LL  &AUB)
{ NodePointer          ptrA, ptrB;
        if  ((AUB = CreateNode()) == NULL) return 0;
        ptrA = A->Next;
        while (ptrA)
         { if (!AddElement(AUB, ptrA->Data)) return 0;
           ptrA = ptrA->Next;
         }
        ptrB = B->Next;
        while (ptrB)
        { if  (!IsAMember(A, ptrB->Data))
                  if (!AddElement(AUB, ptrB->Data)) return 0;
          ptrB = ptrB->Next;
        }
        return 1;
}
```

Tương tự, ta có thể cài đặt các phép toán tập hợp còn lại như: giao, hiệu, hiệu đối xứng, các quan hệ giữa hai tập hợp, ...

c. Biểu diễn đa thức rời rạc bằng DSLK (có nút câm)

Xét đa thức bậc n ($a_n \neq 0$):

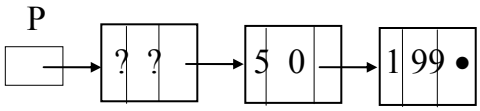
$$P(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

Ta có thể biểu diễn đa thức trên bằng mảng a[n+1] để lưu các hệ số: a[i] = a_i, $\forall$ i = 0, …, n. Với cách biểu diễn này, các phép toán trên đa thức sẽ được thực hiện đơn giản và nhanh chóng. Trong trường hợp *đa thức rời rạc* (đa thức có rất *ít hệ số khác 0),* cài đặt mảng *không hiệu qủa* vì *rất lãng phí bộ nhớ.*

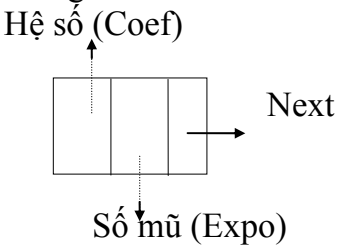
Một cách tiếp cận khác là *dùng DSLK với nút câm để cài đặt đa thức rời rạc*.

* Ví dụ: Xét đa thức

$$P(x) = 5 + x^{99}$$
$$= 5 + 0*x + 0*x^2 + ... + 0*x^{98} + 1*x^{99}$$



Mỗi nút có dạng:



- *Cài đặt đa thức rời rạc*

Trường *dữ liệu Data* của mỗi nút được biểu diễn bởi:

```
typedef   double  CoefType;
typedef   int  ExpoType;
typedef struct { CoefType Coef;
                 ExpoType Expo;
               } ElementType;
```

- *Thủ tục Attach thêm một số hạng x ≡ {x.Coef, x.Expo} vào cuối đa thức P*

```
int  Attach(LL  P , ElementType x)
{
        return  InsertElementTailLL2(P, x);
}
```

- *Thủ tục cộng hai đa thức*

Giả sử các số hạng của các đa thức được lưu tăng theo số mũ vào DSLK đơn có nút câm.

```
int  AddPolynome (LL  A, LL  B, LL  &A_PLUS_B)
{ NodePointer  RestList, ptrA, ptrB;
  CoefType Sum;
  ElementType TempData;
```

```
    if ((A_PLUS_B = CreateEmptyLL2 ( )) == NULL) return 0;
    ptrA = A->Next;
    ptrB = B->Next;
    while (ptrA  &&  ptrB)
      {       if ((ptrA->Data).Expo  <  (ptrB->Data).Expo)
                  {   if (!Attach(A_PLUS_B, ptrA->Data)) return 0;
                      ptrA = ptrA->Next;
                  }
            else  if ((ptrA->Data).Expo  >  (ptrB->Data).Expo)
                                { if (!Attach(A_PLUS_B, ptrB->Data)) return 0;
                                  ptrB = ptrB->Next;
                                }
                  else { TempData.Coef = (ptrA->Data).Coef + (ptrB->Data).Coef;
                         if  (TempData.Coef  != 0) //chỉ lưu các số hạng có hệ số khác 0
                                        { TempData.Expo = ptrtA->Expo;
                                          if (!Attach(A_PLUS_B, TempData)) return 0;
                                        }
                         ptrA = ptrA->Next;
                         ptrB = ptrB->Next;
                       }
      }
    RestList = ptrA;
    if (RestList) RestList = ptrB; // Temp chỉ đến đa thức còn lại có thể chưa hết
    while (RestList)
    { if (!Attach(A_PLUS_B, RestList ->Data)) return 0;
      RestList = RestList ->Next;
    }
    return 1;
}
```

Các thao tác cơ bản khác như: trừ, nhân hai đa thức, lấy thương và phần dư trong phép chia hai đa thức, … được xem như bài tập.

d. Biểu diễn ma trận thưa nhờ DSLK

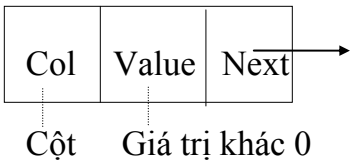
Thông thường ta cài đặt ma trận cấp *m* x *n* bằng mảng 2 chiều. Nhưng trong các bài toán thực tế (chẳng hạn các bài toán trong kết cấu xây dựng, kinh tế, ...) ta thường gặp các *ma trận thưa* (ma trận có *rất ít phần tử khác 0*) có *cấp rất lớn*, *cách cài đặt bởi mảng sẽ không hiệu qủa vì lãng phí bộ nhớ (*thậm chí còn *không khả thi về tốc độ thực hiện khi phải thao tác và lưu trữ những mảng cực lớn trên bộ nhớ phụ),* do phải chứa quá nhiều các phần tử 0 không chứa đựng nhiều thông tin đặc trưng của bài toán. Do đó, cần chọn một kiểu cài đặt khác sao cho *chỉ cần lưu lại các phần tử khác 0* của ma trận.

* <u>*Ví dụ*</u>: Cho ma trận thưa

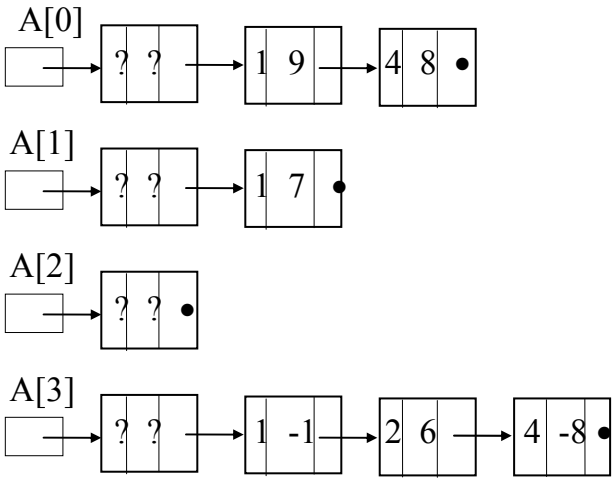
$$A = \begin{bmatrix} 9 & 0 & 0 & 8 & 0 \\ 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

-1 6 0 -8 0

Một trong những cách cài đặt là dùng mảng 1 chiều A[m], trong đó mỗi hàng A[i] là một DSLK chỉ chứa các phần tử khác 0 của hàng thứ i+1 của ma trận, ∀ i = 0 .. m-1. Mỗi nút của DSLK có cấu trúc:



Từ đó, ta có :



- ***Cài đặt cấu trúc dữ liệu cho ma trận thưa***

```
// m là số dòng của ma trận
#define  m  ...
typedef  double ElememtType;  // Kiểu của phần tử của ma trận
typedef NodeType *NodePointer;
typedef struct Node {unsigned  int  Col;
                     ElementType  Value;
                     NodePointer  Next;
                    } NodeType;
NodePointer  PointerArray[m];
PointerArray A;
```

Đối với ma trận có rất nhiều dòng bằng 0, cần phải thay đổi cách cài đặt cho ma trận thưa để việc lưu trữ và các thao tác trên ma trận thưa có hiệu quả hơn bằng cách sử dụng kiểu ***DSLK tổng quát***, nghĩa là *DSLK mà mỗi nút có thể lại là một kiểu DSLK nào đó*. Kiểu DSLK này còn được ứng dụng trong lý thuyết đồ thị, trí tuệ nhân tạo, …

Sau đây, ta minh họa một ứng dụng của DSLK tổng quát vào bài toán sắp xếp tôpô sau đây. Qua đó ta càng thấy rõ tính linh hoạt của kiểu DSLK động.

e. Sắp xếp tôpô

Bài toán *sắp xếp tôppô* dùng để sắp xếp dãy các đối tượng của tập *S* gồm *hữu hạn* phần tử, trên đó có một *quan hệ "thứ tự bộ phận"* $\prec$ thỏa 3 tính chất sau:

1. *Nếu* $x \prec y$ và $y \prec z$ *thì* $x \prec z$ (tính *bắc cầu)*
2. *Nếu* $x \prec y$ *thì không thể có* $y \prec x$ (tính *không đối xứng)*
3. *Không thể có* $x \prec x$ (tính không phản xạ)

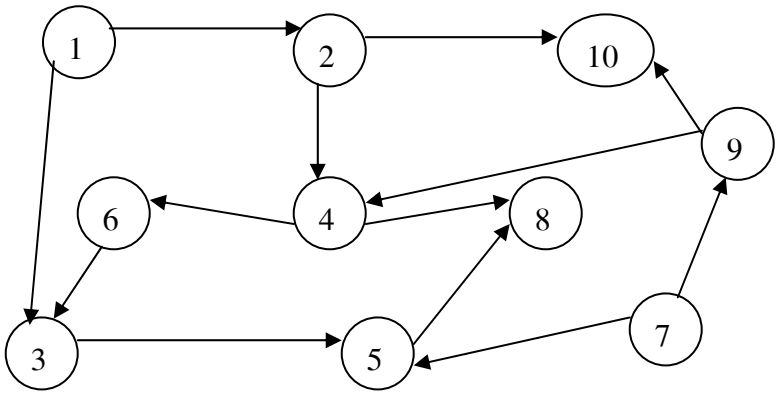
Ta có thể biểu diễn tập S như thế bằng một đồ thị định hướng, không có chu trình (do hai tính chất đầu ở trên), trong đó mỗi đỉnh là một phần tử của S và có một cung nối từ x đến y nếu x, y thỏa quan hệ $\prec$: $x \prec y$.

- ***Bài toán sắp thứ tự tôpô***: là *đưa thứ tự bộ phận về thứ tự tuyến tính*; hay *sắp xếp các đỉnh của đồ thị thành một hàng sao cho tất cả các mũi tên nối các cung đều hướng sang phải*.

Điều kiện *đồ thị không có chu trình* bảo đảm *đưa thứ tự bộ phận về được thứ tự tuyến tính*.

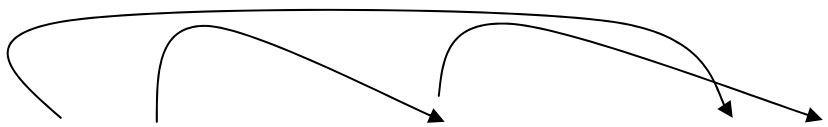
Bài toán trên có nhiều ứng dụng trong thực tế. Chẳng hạn, khi quản lý một đề án nào đó, một công việc lớn thường được chia thành nhiều công việc nhỏ. Thông thường, một việc nhỏ nào đó cần phải được hoàn thành trước các công việc nhỏ khác. Nếu việc v phải xong trước w, ta ký hiệu v $\propto$ w. Sắp xếp tôpô là tổ chức lịch trình thực hiện các công việc sao cho khi thực hiện một công việc nào đó thì mọi việc mà công việc này cần đều phải đã hoàn thành.

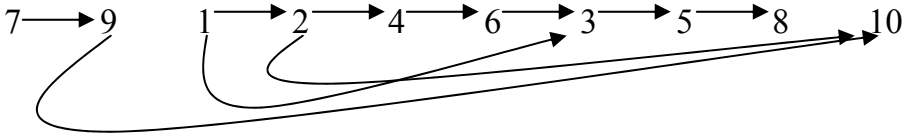
* <u>*Ví dụ*</u>: Sắp xếp tôpô một tập có quan hệ thứ tự bộ phận được biểu diễn bởi đồ thị sau:



hoặc được cho bởi dãy các cặp phần tử sau: 1 $\propto$ 2, 2 $\propto$ 4, 4 $\propto$ 6, 2 $\propto$ 10, 4 $\propto$ 8, 6 $\propto$ 3, 1 $\propto$ 3, 3 $\propto$ 5, 5 $\propto$ 8, 7 $\propto$ 5, 7 $\propto$ 9, 9 $\propto$ 4, 9 $\propto$ 10.

Ta sẽ được (không nhất thiết duy nhất) dãy thứ tự tuyến tính:





- ***Cài đặt cấu trúc dữ liệu:*** Mỗi phần tử của tập được biểu diễn bởi cấu trúc:

```
typedef  int KieuPTu;
typedef struct Leader
            {  KieuPTu key;
               int  count;
                struct Leader *next;
                struct Trailer *trail;
            } LeaderType;
typedef struct Trailer
            {  struct Leader *id;
               struct Trailer *next;
            } TrailerType;
typedef LeaderType *LRef;
typedef TrailerType *TRef;
typedef struct { LRef head, tail;
                 }  LL;
LL  leaders;
```

trong đó: tập các phần tử được lưu trong DSLK *leaders* kiểu LRef; trường *count* dùng để đếm số phần tử đứng trước *key*; trường *trail* dùng để lưu *địa chỉ phần tử đầu* của dãy các *địa chỉ id* của các nút chứa các phần tử đứng sau *key*; *dãy các địa chỉ* này được lưu trong DSLK kiểu *TRef*;

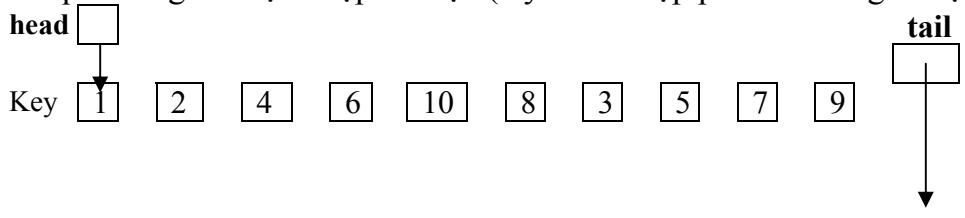
- ***Thuật toán***

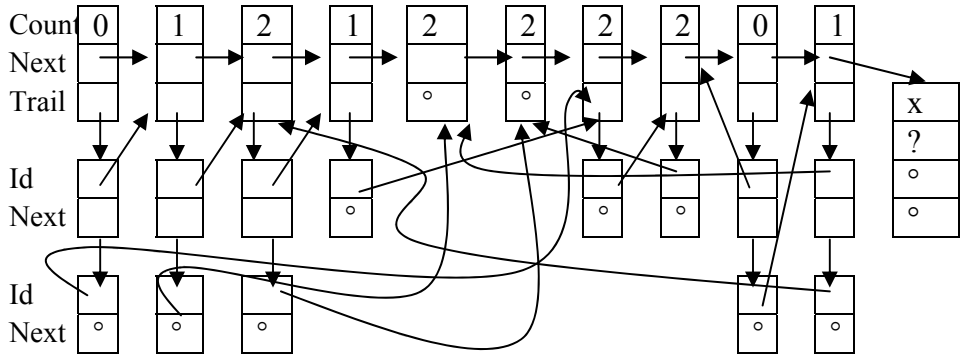
Ý tưởng: Bắt đầu *chọn một phần tử bất kỳ mà không có phần tử nào đứng trước nó* (luôn chọn được vì đồ thị không có chu trình). Tập còn lại, sau khi *loại phần tử này*, vẫn có thứ tự bộ phận và ta *tiếp tục áp dụng cách chọn này* cho đến khi tập trở thành rỗng.

Thuật toán gồm 3 giai đoạn:

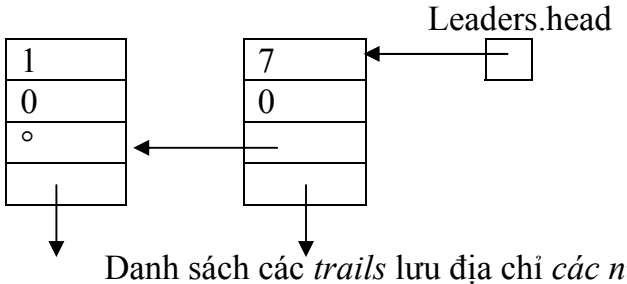
- <u>*Giai đoạn 1*</u>: giai đoạn *nhập*. Lặp lại việc đọc các cặp phần tử của tập S thỏa quan hệ $\prec$ và chèn nó vào DSLK *leaders*, cũng như cập nhật lại các trường đếm số phần tử đứng trước một nút và thêm vào DSLK (kiểu *trail)* các nút chỉ đến nút đứng sau của một nút.

Ta có kết quả của giai đoạn nhập dữ liệu (lấy từ các cặp phần tử trong ví dụ trên)





- <u>*Giai đoạn 2*</u>: *tạo (chẳng hạn, chèn vào đầu) DSLK chứa các phần tử mà chúng không có phần tử nào đứng trước* (cũng gọi là *leaders*, được tạo ra theo thứ tự ngược).
  Chẳng hạn, với ví dụ trên, ta có:



Danh sách các *trails* lưu địa chỉ *các nút đứng sau 7*

- <u>*Giai đoạn 3*</u>: giai đoạn *xuất* các dãy con có thứ tự bộ phận. Dựa vào leaders ở giai đoạn 2, duyệt từng nút q: *xuất (lấy ra khỏi leaders) và giảm đi 1 đơn vị cho trường count* của mọi nút đứng sau q; nếu q->count == 0 thì chèn q vào đầu danh sách leaders.

- ***Cài đặt***

```
void TopoSortLL()
{ LL leaders;
  int  SoPTu = 0;//số phần tử của DS leaders
        NhapDayCapVaoDSach (leaders, SoPTu);
        TachDSCacPTuBatDau(leaders);
        TopoSort(leaders,SoPTu);
        return;
}

int NhapDayCapVaoDSach (LL & leaders, int &SoPTu)
{ KieuPTu x, y;
 LRef p, q;
 TRef t;
        leaders = CreateEmptyLL2();
        while (Nhap1PTu(x))
        { Nhap1PTu(y);
          p = TimChen(x, leaders, SoPTu);
```

```
            q = TimChen(y, leaders, SoPTu);
            t = CreateTrailer();
            if (p && q && t)
                { t->next = p->trail; p->trail = t; // chèn t vào đầu dãy con p->trail của p
                  t->id = q;  // t trỏ đến nút q chứa phần tử đứng sau phần tử trên nút p
                 q->count ++;
                }
            else return 0;
          }
          return 1;
}

int TachDSCacPTuBatDau(List & leaders)
{ LRef p, q;
          p = leaders.head;
          leaders.head = NULL;
          while (p != leaders.tail)
          { q = p; p = p->next;
            if (q->count == 0)
            { q->next = leaders.head;
              leaders.head = q;        // chèn q vào đầu DS leaders
            }
          }
          return 1;
}

int TopoSort(List &leaders, int &SoPTu)
{ LRef p, q = leaders.head;
  TRef t;
          while (q)
          { cout << q->key << '\t';  SoPTu --;
            t = q->trail; q = q->next;
            while (t)
            { p = t->id; p->count --;
              if (p->count  == 0) //Chen p vao ds q
              { p->next = q;  q = p;
              }
              t = t->next;
            }
          }
          if (SoPTu)
              {     cout << "\nTập này không được sắp bộ phận !";  return 0;
              }
          return 1;
}

LRef TimChen(KieuPTu w, List &leaders, int &SoPTu)
{ LRef h = leaders.head;
          (leaders.tail)->key = w; // lưu lính canh ở cuối
```

```
        (leaders.tail)->next = NULL; (leaders.tail)->trail = NULL;
        while (h->key != w) h = h->next;
        if (h == leaders.tail) //khong co phan tu co khoa trong DS leaders
        { if ((leaders.tail = CreateLeader()) == NULL) return NULL;
          SoPTu ++;
          h->count = 0; h->trail = NULL; h->next = ds.tail;
        }
        return h;
}

#define THOAT  0
int Nhap1PTu(KieuPTu &x)
{
        cout << "Nhap 1 ptu:"; cin >> x;
        if (x==THOAT) return 0;
        else return 1;
}
```

Như vậy, chúng ta đã làm quen với hai dạng đơn giản của cấu trúc dữ liệu động là DSLK và cây nhị phân với nhiều cách biểu diễn và cài đặt, cũng như các thao tác cơ bản và một số ứng dụng của chúng. Các phương pháp tìm kiếm và sắp xếp đã được giới thiệu trên cấu trúc mảng tĩnh, DSLK động cũng như cấu trúc cây nhị phân.

Chương IV

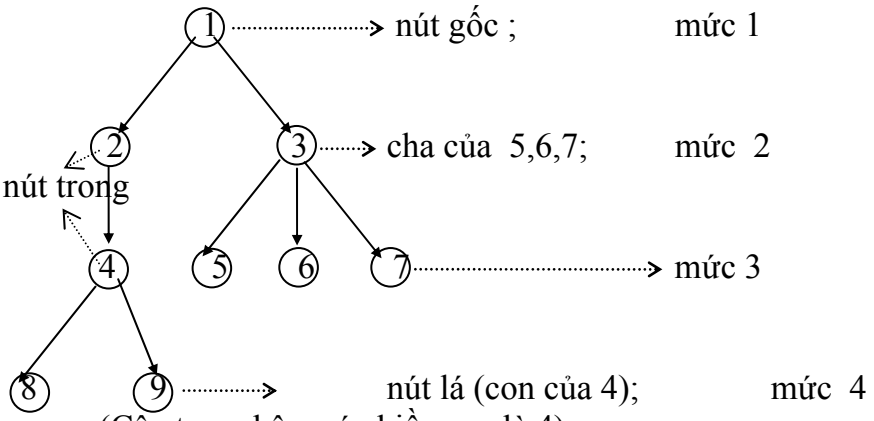
CẤU TRÚC CÂY

Trong cấu trúc dữ liệu động được tổ chức theo kiểu *tuần tự* như danh sách liên kết, tuy có ưu điểm trong các thao tác *chèn, xóa*, nhưng *tốc độ* thực hiện trong các thao tác *truy cập* đến các phần tử của nó hay *tìm kiếm* thường rất *chậm*. Để khắc phục các nhược điểm trên nhưng vẫn duy trì các ưu điểm của cấu trúc dữ liệu động trong các thao tác chèn, xóa, ta có thể dùng một cấu trúc dữ liệu động khác là *cây tìm kiếm* được xét trong chương này để lưu trữ và khai thác dữ liệu hiệu quả hơn.

IV.1. Định nghĩa và các khái niệm cơ bản

IV.1.1. Định nghĩa cây

Cây là một tập hợp ***N*** các phần tử gọi là *nút* (hay đỉnh), trong đó có duy nhất một đỉnh đặc biệt gọi là *gốc,* và một tập hợp các *cạnh có hướng* ***A*** *(A ⊂ NxN)* nối các cặp nút với nhau gọi là *cung* hay *nhánh*. Mỗi nút trên cây đều được nối với gốc bằng duy nhất một dãy các cặp cung liên liếp.



(Cây tam phân, có chiều cao là 4)

Bậc của nút 1 là 2, bậc của nút 2 là 1, bậc của nút 3 là 3, bậc của nút 8 là 0.

IV.1.2. Các khái niệm khác

* Mỗi cung $a_i = (n_i , n_{i+1}) \in A$ có hai nút ở đầu, nút trên n_i gọi là *cha*, nút dưới n_{i+1} gọi là *con*.

* *Nút gốc* là nút (*duy nhất*) không có nút cha. *Mọi nút khác có đúng một nút cha*.

* Một *đường đi* p từ n_1 đến n_k là một dãy các đỉnh $\{n_1, n_2, \dots , n_k\}$ sao cho:

$$a_i = (n_i , n_{i+1}) \in \boldsymbol{A}, \forall\ i = 1, .. , k-1$$

* *Độ dài đường đi* $L_{x,y}$ từ x đến y là số cung trên đường đi từ x đến y. Ký hiệu L_x là độ dài đường đi từ gốc đến x.

* *Độ dài đường đi trung bình* của cây là:

$(\sum_{x \in N} L_x)/n$, n là số nút của cây hay số phần tử của N

trong đó, L_x là độ dài đường đi từ gốc đến đỉnh x.

* Mọi nút khác gốc được *nối với gốc bằng một đường đi duy nhất* bắt đầu từ gốc và kết thúc ở nút đó. Trong cây không có chu trình.

* *Bậc của nút* là số cây con của nút đó.

* *Bậc của cây* là bậc lớn nhất của các nút của cây. Cây bậc n gọi là *cây n - phân*.

* *Nút trong* là nút có bậc lớn hơn không. *Nút lá* là nút có bậc bằng không. Mỗi nút trong cùng với các con của nó tạo thành *cây con*.

* *Mức của 1 nút* (khác nút gốc) là số đỉnh trên đường đi từ gốc đến nút đó. Mức của nút gốc bằng 1:

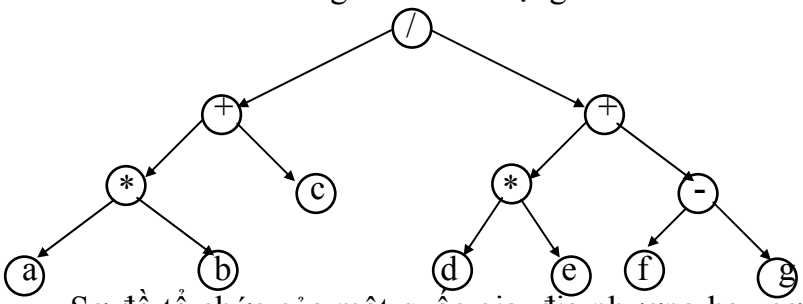
Mức(gốc) = 1;

Mức(con) = Mức(cha) + 1, $\forall$ (cha,con) $\in$ A

* *Chiều cao* của một cây là mức lớn nhất của các nút lá.

* *Ví dụ*: cây có nhiều ứng dụng để biểu diễn các loại dữ liệu trong thực tế. Chẳng hạn:

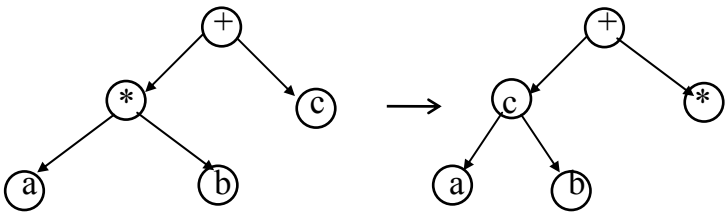
- Biểu thức số học: ((a*b)+c)/((d*e)+(f-g)) được biểu diễn dưới dạng cây. Ta biểu diễn: toán tử bởi nút gốc và toán hạng bởi nút lá.



- Sơ đồ tổ chức của một quốc gia, địa phương hay cơ quan cũng có dạng cây.
- Mục lục sách theo hệ thống phân loại nào đó, …

* *Cây có thứ tự* : là cây mà các nút của nó được xếp theo thứ tự nào đó và có để ý đến vị trí (thứ tự) của các nút con.

Trong cây có thứ tự khi ta thay đổi vị trí của các cây con thì ta sẽ có một cây mới. Chẳng hạn, hai cây có thứ tự sau đây được xem là khác nhau:



* *Cây nhị phân:* là cây mà mỗi nút có *tối đa 2 nút con* (con trái và con phải; do phân biệt vị trí các nút nên cây nhị phân được xem là cây có thứ tự).

* Từ một cây có tổng quát (cây n- phân) ta có thể chuyển về cây nhị phân (xem II.6.) nghĩa là *có thể dùng cây nhị phân để biểu diễn cây tổng quát*. Do tính chất đơn giản và tầm quan trọng như vậy, trước hết ta khảo sát cây nhị phân.

IV.2. Cây nhị phân

IV.2.1. Định nghĩa: *cây nhị phân* là cây (có thứ tự) mà số lớn nhất các *nút con* của các nút là *2*.

Ta còn có thể xem cây nhị phân như là một cấu trúc dữ liệu đệ qui.

* *Định nghĩa đệ qui*: Một *cây nhị phân (Binary tree) :*

+ *hoặc là rỗng* (phần neo hay trường hợp cơ sở);
+ *hoặc là một nút* mà nó *có 2 cây con nhị phân không giao nhau*, gọi là cây con bên trái và cây con bên phải (phần đệ qui).

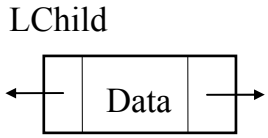
IV.2.2. Vài tính chất của cây nhị phân

Gọi *h* và *n* lần lượt là chiều cao và số phần tử của cây nhị phân.

- Số nút ở mức i $\leq 2^{i-1}$, hay nói chính xác hơn *số nút tối đa ở mức i là* 2^{i-1}. Do đó, số nút lá tối đa của nó là 2^{h-1}.
- Số nút tối đa trong cây nhị phân là $2^h - 1$, hay $n \leq 2^h - 1$. Do đó, chiều cao của nó: $n \geq h \geq \log_2(n+1)$

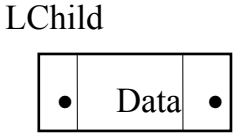
IV.2.3. Biểu diễn cây nhị phân

Ta chọn cấu trúc động để biểu diễn *mỗi nút* trên cây nhị phân:



trong đó: LChild, RChild lần lượt là các con trỏ chỉ đến nút con bên trái và nút con phải. LChild hay RChild là con trỏ rỗng nếu không có nút con bên trái hay bên phải.

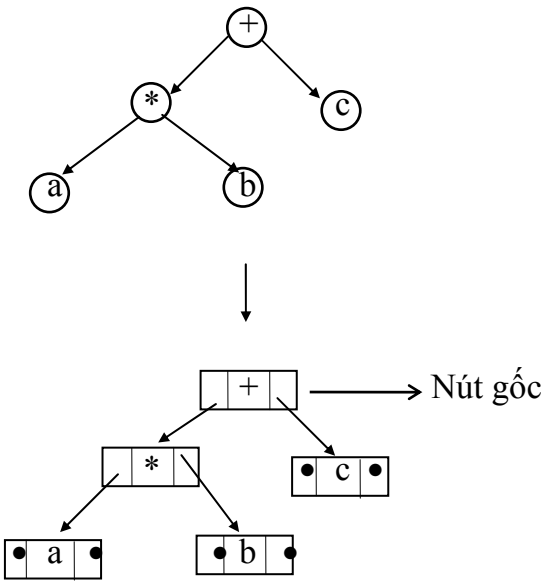
Nút lá có dạng:



Trong ngôn ngữ C hay C++, ta *khai báo kiểu dữ liệu cho một nút* của cây nhị phân như sau:

```
typedef    .....   ElementType;  /* Kiểu mục dữ liệu của nút */
typedef struct TN { ElementType Data; //Để đơn giản, ta xem Data là trường khóa của dữ liệu
                    struct TN * LChild, *RChild;
                  } TreeNode;
typedef   TreeNode  *TreePointer;
```

* *Ví dụ*: Ta biểu diễn biểu thức số học: a * b + c bởi cây nhị phân:



Trong các thuật toán thuộc chương này, ta sẽ sử dụng hàm CấpPhát() để cấp phát vùng nhớ cho một nút mới của cây nhị phân. Hàm trả về địa chỉ bắt đầu vùng nhớ được cấp phát cho một nút nếu việc cấp phát thành công và trả trị NULL nếu ngược lại. Trong C++, hàm trên có thể được viết như sau:

TreePointer CấpPhát ()

```
{TreePointer Tam= new TreeNode;
        if (Tam == NULL)
                cout << "\nLỗi cấp phát vùng nhớ cho một nút mới của cây nhị phân !";
        return Tam;
}
```

IV.2.4. Duyệt cây nhị phân

IV.2.4.1. Định nghĩa*: Duyệt qua cây nhị phân là quét qua mọi nút* của *cây nhị phân* sao cho *mỗi nút được xử lý đúng một lần.*

Dựa vào định nghĩa đệ qui ta chia cây nhị phân ra làm 3 phần: gốc, cây con bên trái, cây con bên phải. Ta có *3 phương pháp chính duyệt cây nhị phân tùy theo trình tự duyệt 3 phần trên*:

+ Duyệt qua theo thứ tự giữa (LNR)

+ Duyệt qua theo thứ tự đầu (NLR)
+ Duyệt qua theo thứ tự cuối (LRN).

trong đó:

L : quét cây con trái của một nút
R : quét cây con phải của một nút
N : xử lý nút.

IV.2.4.2. Các thuật toán duyệt cây nhị phân

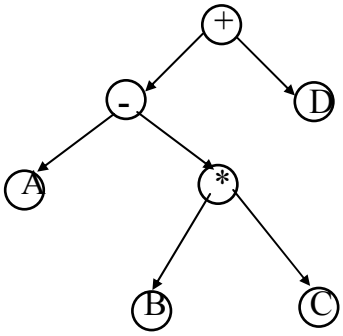
* *Thuật toán duyệt qua theo thứ tự giữa (LNR: Trái - Gốc - Phải)* :
+Duyệt qua cây con trái theo thứ tự giữa;
+Duyệt qua gốc;
+Duyệt qua cây con phải theo thứ tự giữa.

* *Thuật toán duyệt qua theo thứ tự đầu (NLR: Gốc - Trái - Phải):*
+Duyệt qua gốc;
+Duyệt qua cây con trái theo thứ tự đầu;
+Duyệt qua cây con phải thứ tự đầu.

Thuật toán NLR sẽ *duyệt cây theo chiều sâu*.

* *Thuật toán duyệt qua theo thứ tự cuối* (LRN: Trái - Phải - Gốc):
+Duyệt qua cây con trái theo thứ tự cuối;
+Duyệt qua cây con phải theo thứ tự cuối;
+Duyệt qua gốc.

* *Ví dụ*: Biểu diễn biểu thức: A - B * C + D lên cây nhị phân:



Duyệt cây theo các thứ tự khác nhau:
LNR: A - B * C + D (biểu thức *trung tố*)
NLR: + - A * B C D (biểu thức *tiền tố*)
LRN: A B C * - D + (biểu thức *hậu tố*)

Với cách biểu diễn một biểu thức số học dưới dạng cây nhị phân, dựa trên cách duyệt *LRN* ta có thể *tính giá trị của biểu thức* đó (*Bài tập*).

Do định nghĩa đệ quy của cây nhị phân, các thuật toán duyệt qua cây theo kiểu đệ quy là thích hợp.

IV.2.4.3. Cài đặt thuật toán duyệt qua cây nhị phân LNR

a. Cài đặt thuật toán LNR dưới dạng đệ qui :

```
/* Input: - Root : con trỏ chỉ đến nút gốc của cây nhị phân
   Output: - Duyệt qua và xử lý mọi nút của cây nhị phân theo thứ tự giữa LNR
*/
void LNRĐệQuy (TreePointer Root)
{       if (Root != NULL)
        {       LNRĐệQuy (Root->LChild);
                Xử lý (Root); //Xử lý theo yêu cầu cụ thể, chẳng hạn: Xuất(Root->Data);
                LNRĐệQuy (Root->RChild) ;
        }
        return;
}
```

Thuật toán duyệt cây nhị phân theo thứ tự giữa (LNR) có thể viết lại dưới dạng lặp, bằng cách sử dụng một *stack* để *lưu lại địa chỉ các nút gốc trước khi đi đến cây con trái* của nó. Trước hết, ta khai báo cấu trúc một nút của *stack* trên:

```
typedef   struct  NS { TreePointer  Data;
                       struct NS * Next;
                     } NodeStack;
typedef   NodeStack  * StackType;
```

b. Cài đặt thuật toán LNR dưới dạng lặp :

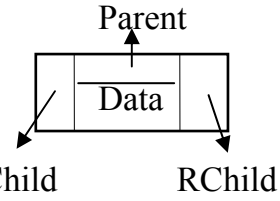
```
/* Input:   - Root : con trỏ chỉ đến nút gốc của cây nhị phân
   Output: - Duyệt qua và xử lý mọi nút của cây nhị phân theo thứ tự giữa LNR
*/

void  LNRLap(TreePointer Root)
{ TreePointer p;
   int TiepTuc = 1;
  StackType S;
        p = Root; S = CreateEmptyStack();         // Khởi tạo ngăn xếp rỗng
        do
        { while (p != NULL)
                { Push(S,p);                      // Đẩy p vào stack S
                  p = p->LChild;
                }
          if (!EmptyStack(S))                     // Nếu stack S khác rỗng
                { Pop(S,p);               // Lấy ra phần tử p ở đỉnh stack S
                  XuLy(p);
                  p =  p->RChild;
                }
```

```
        else TiepTuc = 0;
    } while (TiepTuc);
    return ;
}
```

Với hai trường hợp duyệt cây còn lại (NLR và LRN), ta cũng có thể cài đặt chúng dưới dạng đệ quy và lặp (*bài tập*). Một cách tổng quát, ta có thể viết lại ba thuật toán duyệt này dưới một dạng lặp duy nhất (*bài tập*).

IV.2.5. Một cách biểu diễn khác của cây nhị phân

Trong một số trường hợp, khi biểu diễn cây nhị phân, người ta không chỉ quan tâm đến quan hệ một chiều từ cha đến con mà cả chiều ngược lại: từ con đến cha. Khi đó, ta có thể dùng cấu trúc sau:

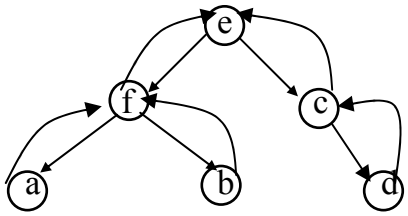


trong đó: LChild, RChild lần lượt là các con trỏ chỉ đến nút con trái và nút con phải. Parent là con trỏ chỉ đến nút cha.

Trong ngôn ngữ C hay C++, ta *khai báo kiểu dữ liệu cho một nút* của cây nhị phân dạng này như sau:

```
typedef    .....   ElementType;  /* Kiểu mục dữ liệu của nút */
typedef struct TNP {ElementType Data; //Để đơn giản, ta xem Data là trường khóa của dữ liệu
                    struct TNP   * LChild, *Rchild, *Parent;
                   } TreeNodeP;
typedef   TreeNodeP  *TreePointer;
```

* *Ví dụ*:



IV.2.6. Biểu diễn cây n - phân bởi cây nhị phân.

Phương pháp *cài đặt cây n - phân* bằng mảng *có n vùng liên kết* chỉ có *lợi khi hầu hết các nút của cây có bậc là n*. Khi đó n vùng liên kết đều được sử dụng,

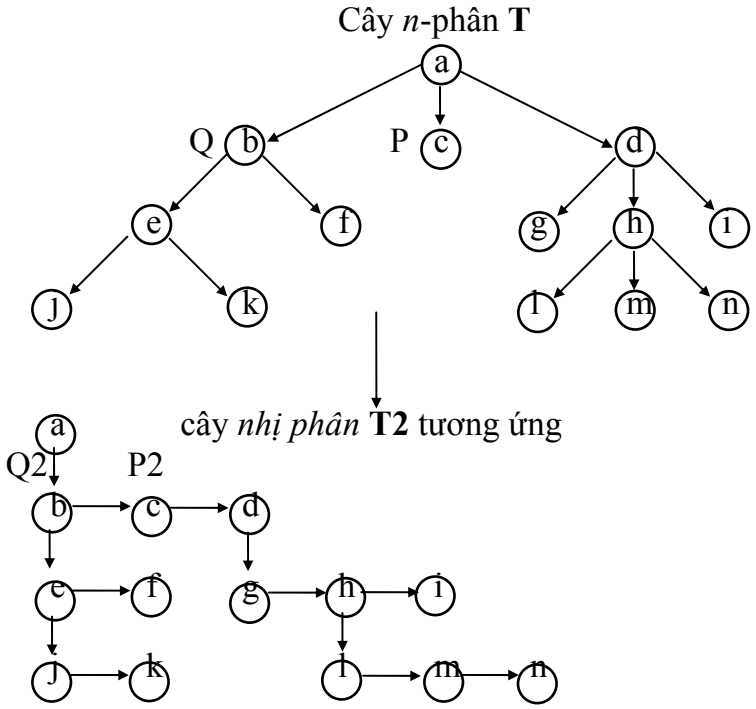
nhưng với cây có nhiều nút có bậc nhỏ hơn *n* sẽ gây nên việc *lãng phí bộ nhớ* vì có nhiều vùng liên kết không sử dụng tới.

Do cây nhị phân là cấu trúc dữ liệu cây cơ bản và đơn giản đã được nghiên cứu, nên để mô tả cây n-phân, người ta tìm cách biểu diễn nó thông qua cây nhị phân. Gọi: T là cây n-phân, T2 là cây nhị phân tương ứng với T. Ta gọi các nút con của cùng một nút là *anh em* với nhau. Để biểu diễn T bằng T2, ta theo các qui tắc sau:

+ Nút gốc trong T được biểu diễn tương ứng với nút gốc của T2.

+ Con đầu tiên (trái nhất) của một nút trong T là con trái của nút tương ứng trong T2.

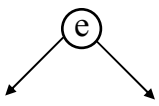
+ Nút anh em kề phải P của một nút Q trong T tương ứng với một nút P2 trong T2 qua liên kết phải của nút Q2 tương ứng trong T2.

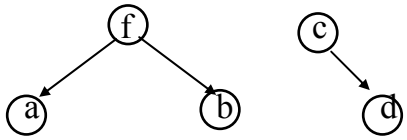


IV.2.7. Xây dựng cây nhị phân cân bằng hoàn toàn

IV.2.7.1. <u>*Định nghĩa*</u>: Cây nhị phân cân bằng hoàn toàn (CBHT) là cây nhị phân mà đối với mỗi nút của nó, số nút của cây con trái chênh lệch không quá 1 so với số nút của cây con phải.

* <u>*Ví dụ*</u>:





IV.2.7.2. Xây dựng cây nhị phân cân bằng hoàn toàn

Xây dựng cây nhị phân cân bằng hoàn toàn có *n* phần tử:

```
TreePointer  TạoCâyCBHT(Nguyên n)
{ TreePointer Root;
  Nguyên nl, nr;
  ElementType x;
        if (n<=0) return NULL;
        nl = n/2; nr = n-nl-1;
        Nhập1PhầnTử(x);
        if ((Root =CấpPhát()) == NULL) return NULL;
        Root->Data = x;
        Root->LChild = TạoCâyCBHT(nl);
        Root->RChild = TạoCâyCBHT(nr);
        return Root;
}
```

* *Nhận xét*:

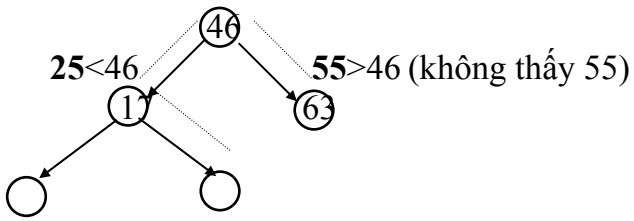
- Một cây CBHT có *n* nút sẽ có chiều cao bé nhất $h \approx log_2 n$.
- Một cây CBHT *rất dễ mất cân bằng* sau khi *thêm* hay *hủy* các nút trên cây, việc *chi phí cân bằng lại cây rất lớn* vì phải *thao tác lại trên toàn bộ cây*. Do đó cây CBHT có cấu trúc kém ổn định, ít được sử dụng trong thực tế.

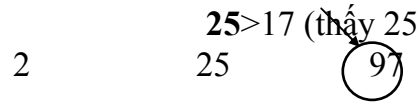
IV.3. Cây nhị phân tìm kiếm (BST)

IV.3.1. Định nghĩa cây nhị phân tìm kiếm (BST)

Cây BST là một cây nhị phân có tính chất giá trị khóa ở mỗi nút lớn hơn giá trị khoá của mọi nút thuộc cây con bên trái (nếu có) và nhỏ hơn giá trị khoá của mọi nút thuộc cây con bên phải (nếu có) của nó.

* *Ví dụ*: Xét cây BST sau đây lưu các giá trị: 46, 17, 63,2, 25, 97. Ta biểu diễn quá trình tìm kiếm 2 phần tử 25, 55 trên cây BST qua hình dưới đây:





Với loại cấu trúc dữ liệu động *danh sách liên kết,* ta rất khó áp dụng hiệu qủa ý tưởng tìm kiếm nhị phân trên mảng. Nhưng với loại cấu trúc dữ liệu động cây BST thì việc thể hiện ý tưởng này là đơn giản.

IV.3.2. Tìm kiếm một phần tử trên cây BST

(Thuật toán tìm kiếm nhị phân sau đây tương tự phép tìm kiếm nhị phân trên mảng).

IV.3.2.1. Thuật toán tìm kiếm dạng đệ qui:

```
/* Input: - Root: con trỏ chỉ đến nút gốc của cây BST.
          - Item: giá trị khóa của phần tử cần tìm .
Output:  - Trả về con trỏ LocPtr chỉ đến 1 nút trên cây BST chứa Item nếu tìm
           thấy Item trên cây BST
         - Trả trị NULL nếu ngược lại */
TreePointer TìmBSTĐệQuy (TreePointer Root, ElementType Item)
{
        if (Root)
              {if (Item== Root->Data) return Root;
               else     if (Item > Root->Data) return TìmBSTĐệQuy (Root-
>RChild,Item);
                        else return TìmBSTĐệQuy (Root->LChild,Item);
              }
        else return(NULL);
}
```

* Thủ tục được viết dưới *dạng đệ qui thích hợp* với lối *tư duy tự nhiên của giải thuật và định nghĩa đệ qui* của cây nhị phân. Song trong trường hợp này thủ tục viết dưới *dạng lặp lại tỏ ra hiệu quả hơn*.

IV.3.2.2. Thuật toán tìm kiếm dạng lặp:

```
/* Input: - Root: con trỏ chỉ đến nút gốc của cây BST.
          - Item: giá trị khóa của phần tử cần tìm .
Output:  - Trả về con trỏ LocPtr chỉ đến 1 nút trên cây BST chứa Item và con trỏ
           Parent chỉ đến nút cha của nút chứa Item đó nếu tìm thấy Item trên cây
           BST
         - Trả trị NULL nếu ngược lại */
TreePointer TìmBSTLặp(TreePointer Root, ElementType Item, TreePointer &Parent)
{ TreePointer LocPtr = Root;
        Parent = NULL;
        while (LocPtr != NULL)
              if (Item==LocPtr->Data)  return (LocPtr);
```

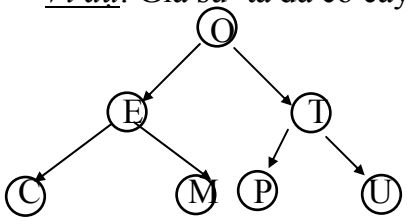
```
            else   {Parent = LocPtr;
                    if (Item > LocPtr->Data) LocPtr = LocPtr->RChild;
                    else  LocPtr = LocPtr->LChild;
                   }
        return(NULL);
}
```

Với cấu trúc cây, việc *tìm kiếm theo khóa sẽ nhanh hơn nhiều* so với cấu trúc danh sách liên kết. Chi phí tìm kiếm (*độ phức tạp) trung bình* trên cây nhị phân có *n* nút khoảng $\boldsymbol{log_2 n}$.

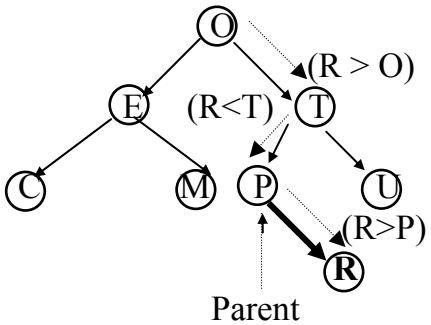
IV.3.3. Chèn một phần tử vào cây BST, xây dựng cây BST

Việc chèn thêm một phần tử Item vào cây BST cần phải thỏa ràng buộc trong định nghĩa cây BST. Trước khi chèn Item, ta cần *tìm* khóa của Item có trong cây BST hay không, nếu có thì khỏi chèn (do trên cây BST ta chỉ chứa những phần tử *có khóa khác nhau*); nếu ngược lại, khi *chấm dứt thao tác tìm kiếm thì ta cũng biết được vị trí chèn* (*ở nút lá*).

* *Ví dụ*: Giả sử ta đã có cây BST (với các nút có khóa khác nhau):



Ta cần thêm phần tử 'R':



Yêu cầu "vào – ra" của thao tác chèn:

```
/* Input: - Root: con trỏ chỉ đến nút gốc của cây BST.
          - Item: giá trị dữ liệu của nút cần chèn
 Output: - Trả trị 1 và con trỏ Root chỉ đến nút gốc mới của cây BST nếu chèn được
          - Trả trị -1 nếu Item đã có trên cây
```

- Trả trị 0 nếu gặp lỗi cấp phát bộ nhớ cho một nút mới của cây */

IV.3.3.1. Thao tác chèn một nút Item vào cây BST (dạng lặp):

```
int ChènBSTLặp(TreePointer &Root, ElementType Item)
{ TreePointer LocPtr, Parent;
    if (TìmBSTLặp(Root, Item, Parent))
        { cout << "\nĐã có phần tử "<< Item << " trong cây !" ;
          return -1;
        }
    else { if ((LocPtr=CấpPhát ())==NULL) return 0;
           LocPtr->Data = Item;
           LocPtr->LChild = NULL; LocPtr->RChild = NULL;
           if (Parent == NULL)
                  Root = LocPtr;   // cây rỗng
           else   if (Item < Parent->Data)  Parent->LChild =  LocPtr;
                  else Parent->RChild = LocPtr;
           return 1;
         }
}
```

IV.3.3.2. Thủ tục chèn một nút Item vào cây BST (dạng đệ qui):

```
int ChènBSTĐệQui(TreePointer &Root, ElementType Item)
{ TreePointer LocPtr;
    if (Root ==  (TreePointer) NULL)          // chèn nút vào cây rỗng
           {  if ((Root = CấpPhát ()) == NULL) return 0;
              Root ->Data = Item;
              Root ->LChild = NULL; Root ->RChild = NULL;
           }
    else   if (Item < Root->Data) ChènBSTĐệQui (Root->LChild,Item);
           else   if (Item > Root->Data) ChènBSTĐệQui(Root->RChild,Item);
                  else { cout <<  "\nĐã có phần tử "<< Item << " trong cây";
                         return -1;
                       }
    return 1;
}
```

IV.3.3.3. Xây dựng cây BST

Ta có thể ***xây dựng cây BST*** bằng cách lặp lại thao tác chèn một phần tử vào cây BST trên đây, xuất phát từ cây rỗng. Hàm *TạoCâyBST(Root)* sau đây trả về trị 0 nếu gặp lỗi cấp phát vùng nhớ cho một nút mới của cây Root và trả về trị 1 nếu việc chèn các nút vào cây thành công (không chèn các nút có khóa đã trùng với khóa của nút đã chèn).

```
int TạoCâyBST(PointerType &Root)
{ ElementType Item;
        Root = NULL;
        while (ConLấyDữLiệu(Item))
                if (!ChènBSTLặp(Root, Item)) return 0;
        return 1;
}
```

IV.3.4. Phương pháp sắp xếp bằng cây BST

Ta nhận xét rằng sau khi duyệt một cây BST theo thứ tự giữa *LNR* thì ta sẽ *thu được một dãy tăng theo khóa*. Từ đó, ta có phương pháp sắp xếp dựa trên cây BST như sau. Giả sử ta cần sắp xếp dãy X các phần tử.

* *Giải thuật* ***BSTSort***:

- Bước 1: Đưa lần lượt mọi phần tử của dãy X lên cây BST.
- Bước 2: Khởi tạo lại dãy rỗng X. Duyệt cây BST theo thứ tự giữa (*LNR*), trong đó thao tác *XửLý*(Nút) lưu *Nút->Data vào phần tử tiếp theo* của dãy *X*.

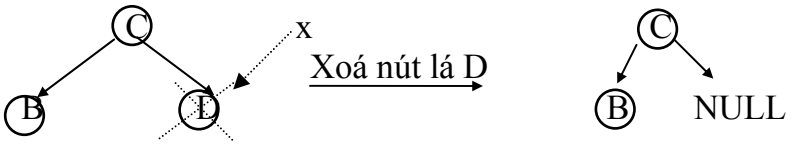
* *Ví dụ*: Giả sử cần sắp xếp một dãy gồm *n* phần tử được lưu trong mảng X. Khi đó ta có thuật toán sau:

1.Khởi tạo cây BST rỗng.
2.for (i = 0; i< n; i++) Chèn X[i] vào cây BST;
3.Đặt lại i = 0;
4.Duyệt qua theo thứ tự giữa LNR, việc XửLý(Nút) một nút khi duyệt qua cây là:
- Gán X[i] ← Nút->Data;
- Tăng i lên 1;

IV.3.5. Xóa một phần tử khỏi cây BST, hủy cây nhị phân

Giả sử ta cần xóa một nút (trên cây BST) được trỏ bởi *x*. Việc xoá một phần tử trên cây BST cũng *cần phải thoả các ràng buộc về cây BST*, nhưng việc xóa phức tạp hơn so với chèn. Ta phân biệt *3* trường hợp : *x* trỏ đến nút *lá*, *x* trỏ đến *nút chỉ có một con*, *x* trỏ đến nút *có hai con*.

a). Xoá nút lá:

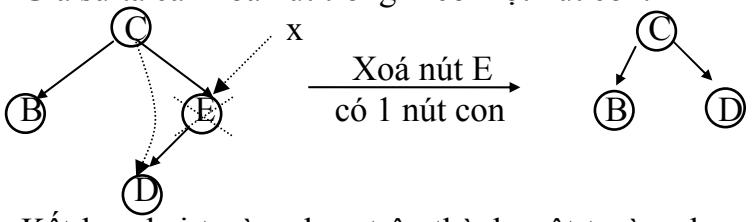


- Đặt con trỏ phải (hay trái) của nút cha của *x* thành NULL
- Giải tỏa nút D

b). <u>Xoá nút có một nút con</u>:

- Đặt con trỏ phải (hoặc trái) của nút cha của nút cần xóa trỏ đến nút con khác rỗng của nút cần xóa
- Giải tỏa nút cần xóa

Giả sử ta cần xóa nút trong E có một nút con:



Kết hợp hai trường hợp trên thành một trường hợp: *x* trỏ đến nút có nhiều nhất một cây con khác rỗng. Gọi:

+ *x* chỉ đến nút cần xóa

+ *SubTree* chỉ đến cây con (khác rỗng , nếu có) của *x*

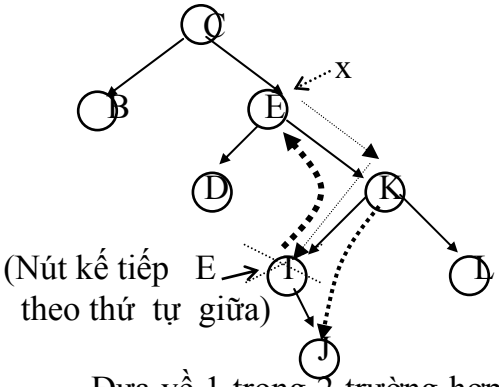
+ *Parent* chỉ đến nút cha của nút được trỏ bởi *x* (nếu *x* chỉ đến gốc thì *Parent*=NULL).

Ta có giải thuật xóa cho trường hợp này là:

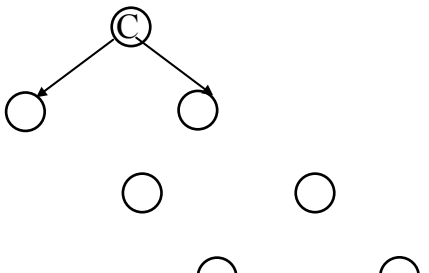
```
SubTree =  x->LChild;
if  (SubTree ==  NULL )  SubTree =  x->RChild;
//SubTree là cây con khác rỗng (nếu có) của x
if  (Parent == NULL)  Root = SubTree;  // xoá nút gốc
else     if  (Parent->LChild == x)   Parent->LChild = SubTree ;
         else   Parent->RChild = SubTree;
delete  x;
```

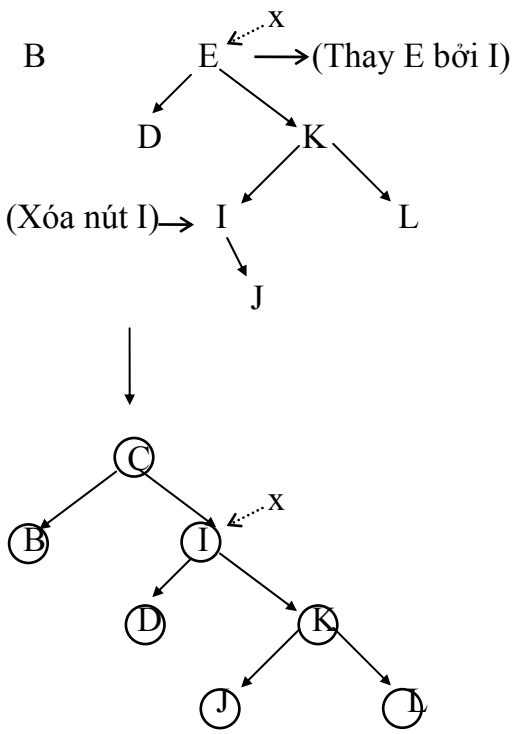
c). <u>Xoá nút có hai nút con</u>:

Giả sử ta cần xoá nút E có 2 nút con của cây BST sau :



Đưa về 1 trong 2 trường hợp đầu bằng cách sau: Thay trị của nút mà *x* trỏ đến bởi trị của nút kế tiếp theo thứ tự giữa (nút kế tiếp là nút cực trái xa nhất theo nhánh con phải của *x*, hay là nút nhỏ nhất (tất nhiên là theo trường khóa) trong số những nút lớn hơn *x->Data*). Sau đó xoá nút kế tiếp này (nút kế tiếp này sẽ là nút có tối đa 1 nút con).





* Sau đây ta xây dựng thủ tục *XóaBST* để xóa một nút *Item* trong một cây BST. Trong thủ tục này có dùng đến thủ tục *TìmBSTLặp*. Thủ tục *XoáBST* tìm nút có khóa *Item* và xoá nó khỏi cây BST.

Gọi:

- *x*: trỏ đến nút chứa *Item*
- *xSucc*: phần tử kế tiếp của *x* theo thứ tự giữa (nếu *x* có 2 con)
- *Parent*: trỏ đến cha của *x* hay *xSucc*
- *SubTree*: trỏ đến cây con của *x*.

/* Input: - Root: con trỏ chỉ đến nút gốc của cây BST.
- Item: giá trị dữ liệu của nút cần xóa
Output: - Trả trị 1 và con trỏ Root chỉ đến nút gốc mới của cây BST nếu tìm thấy nút
chứa Item và xoá được
- Trả trị 0 nếu ngược lại */

int XóaBST (TreePointer &Root, ElementType Item)

```
{ TreePointer x,Parent, xSucc,SubTree;
    if ((x = TìmBSTLặp(Root,Item,Parent)) ==NULL) return 0;//không thấy Item
    else { if ((x->LChild != NULL) && (x->RChild != NULL)) // nút có 2 con
              { xSucc = x->RChild;
                Parent = x;
```

```
                while (xSucc->LChild != NULL)
                 { Parent = xSucc;
                    xSucc =  xSucc->LChild;
                 }
                x->Data = xSucc->Data; x = xSucc;
               }                  //đã đưa nút có 2 con về nút có tối đa 1 con
            SubTree =  x->LChild;
            if  (SubTree ==  NULL) SubTree =  x->RChild;
            if  (Parent == NULL)  Root =  SubTree;          // xoá nút gốc
            else  if   (Parent->LChild == x)  Parent->LChild = SubTree;
                  else  Parent->RChild = SubTree;
            delete  x;
            return 1;
          }
}
```

Ta có thể ***hủy toàn bộ cây BST*** bằng cách sử dụng ý tưởng duyệt cây theo thứ tự cuối LRN: hủy cây con trái, hủy cây con phải rồi mới hủy nút gốc.

```
void HủyCâyNhịPhân (PointerType &Root)
{       if (Root)
        { HủyCâyNhịPhân (Root->LChild);
          HủyCâyNhịPhân (Root->RChild);
           delete Root;
        }
        return ;
}
```

IV.4. Cây nhị phân tìm kiếm cân bằng

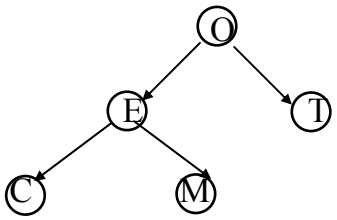
Trên cây *nhị phân tìm kiếm BST* có *n* phần tử mà là cây CBHT (cân bằng hoàn toàn), phép *tìm kiếm* một phần tử trên nó sẽ thực hiện *rất nhanh*: trong trường hợp *xấu nhất*, ta chỉ cần thực hiện $\boldsymbol{log_2 n}$ phép so sánh. Nhưng *cây CBHT có cấu trúc kém ổn định* trong các thao tác cập nhật cây, nên nó ít được sử dụng trong thực tế. Vì thế, người ta tận dụng ý tưởng cây CBHT để xây dựng một cây nhị phân tìm kiếm có trạng thái cân bằng yếu hơn, nhưng việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ đồng thời chi phí cho việc tìm kiếm vẫn dạt ở mức O($\boldsymbol{log_2 n}$). Đó là cây nhị phân tìm kiếm cân bằng.

IV.4.1. Định nghĩa

Cây *nhị phân tìm kiếm* gọi là cây *nhị phân tìm kiếm cân bằng (*gọi tắt là *cây cân bằng* hay *cây AVL* do 3 tác giả *Adelson-Velskii-Landis* đưa ra vào năm 1962*)* nếu tại mỗi nút của nó, *độ cao của cây con trái và độ cao của cây con phải chênh lệch không quá 1*.

Rõ ràng, một cây nhị phân tìm kiếm cân bằng hoàn toàn là cây cân bằng, nhưng điều ngược lại không đúng. Chẳng hạn cây nhị phân tìm kiếm trong ví dụ sau là cân bằng nhưng không phải là cân bằng hoàn toàn:

* *Ví dụ*: (cây nhị phân tìm kiếm cân bằng nhưng không cân bằng hoàn toàn)



Cây cân bằng *AVL vẫn thực hiện việc tìm kiếm nhanh tương đương cây* (nhị phân tìm kiếm) *cân bằng hoàn toàn và vẫn có cấu trúc ổn định hơn* hẳn cây cân bằng hoàn toàn mà nó được thể hiện qua các thao tác cơ bản sẽ được trình bày trong các phần tiếp theo.

IV.4.2. Chiều cao của cây cân bằng

* *Định lý* (AVL): Gọi $h_b(n)$ là độ cao của cây AVL có n nút, khi đó:

$$log_2(n+1) \le h_b(n) < 1.4404 * log_2(n+2) - 0.3277$$

Cây AVL là *tối ưu* (trong trường hợp tốt nhất, nó có *chiều cao bé nhất*) khi nó là cây *cân bằng hoàn toàn* có n nút với: $n = 2^k - 1$. Một *cây AVL không bao giờ cao quá 45% cây cân bằng hoàn toàn* tương ứng của nó.

Chứng minh: Bất đẳng thức thứ nhất ở bên trái có được do tính chất của cây nhị phân (phần II.2).

Để chứng minh bất đẳng thức thứ hai ở bên phải, ta gọi $N(h)$ là số nút ít nhất của cây AVL $T(h)$ có chiều cao h.

Ta có: N(0) = 0 ứng với cây rỗng T(0) và N(1) = 1 ứng với cây chỉ có 1 nút T(1). Khi h > 1, gốc của cây T(h) sẽ có hai cây con cũng có số nút ít nhất, một cây có chiều cao là h -1, cây con kia có chiều cao là h -2. Do đó:

$$\begin{cases} N(h) = 1 + N(h-1) + N(h-2), & \forall\ h > 1 \\ N(0) = 0, \quad N(1) = 1. \end{cases}$$

Đặt F(h) = N(h) + 1. Khi đó:

$$\begin{cases} F(h) = F(h-1) + F(h-2), & \forall\ h > 1 \\ F(0) = 1, \quad F(1) = 2. \end{cases}$$

Giải hệ thức truy hồi trên (*bằng cách nào ? Bài tập*), ta được:

$$n + 1 \ge N(h) + 1 = F(h) = (r_1^{h+2} - r_2^{h+2}) / \sqrt{5} > (r_1^{h+2} - 1) / \sqrt{5}$$

với: $r_1 = (1+\sqrt{5})/2$, $r_2 = (1-\sqrt{5})/2 \in (-1; 1)$

$\Rightarrow h + 2 < \log_{r1}(1+\sqrt{5}(n+1)) < \log_{r1}(\sqrt{5}(n+2)) < \log_{r1}(n+2) + \log_{r1}(\sqrt{5})$

$h < \log_2(n+2)/\log_2(r_1) + \log_{r1}(\sqrt{5}) - 2 \approx 1.44042 \log_2(n+2) - 0.32772$

Vậy một cây AVL có *n* nút sẽ có *chiều cao tối đa (trong trường hợp xấu nhất) là O(log$_2$n).*

IV.4.3. Chỉ số cân bằng và việc cân bằng lại cây AVL

* <u>Định nghĩa</u>: *Chỉ số cân bằng (CSCB)* của một nút *p* là *hiệu* của *chiều cao* cây con *phải và* cây con *trái* của nó.

Ký hiệu: $h_L(p)$ hay h_L là chiều cao cây con trái (của p),

$h_R(p)$ hay h_R là chiều cao cây con phải (của p),

EH = 0, RH = 1, LH = -1.

CSCB(p) = EH ⇔ $h_R(p) = h_L(p)$: 2 cây con cao bằng nhau

CSCB(p) = RH ⇔ $h_R(p) > h_L(p)$: cây lệch phải

CSCB(p) = LH ⇔ $h_R(p) < h_L(p)$: cây lệch trái

Với mỗi nút của *cây AVL*, ngoài các thuộc tính thông thường như cây nhị phân, ta cần lưu thêm thông tin về chỉ số cân bằng trong *cấu trúc của một nút*:

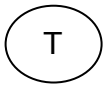
```
typedef  .....  ElementType;  /* Kiểu mục dữ liệu của nút */
typedef  struct  AVLTN { ElementType Data; //Ở đây ta xem Data là trường khóa của dữ liệu
                         int  Balfactor; //Chỉ số cân bằng
                         struct AVLTN * Lchild, *Rchild;
                       } AVLTreeNode;
typedef  AVLTreeNode  *AVLTree;
```

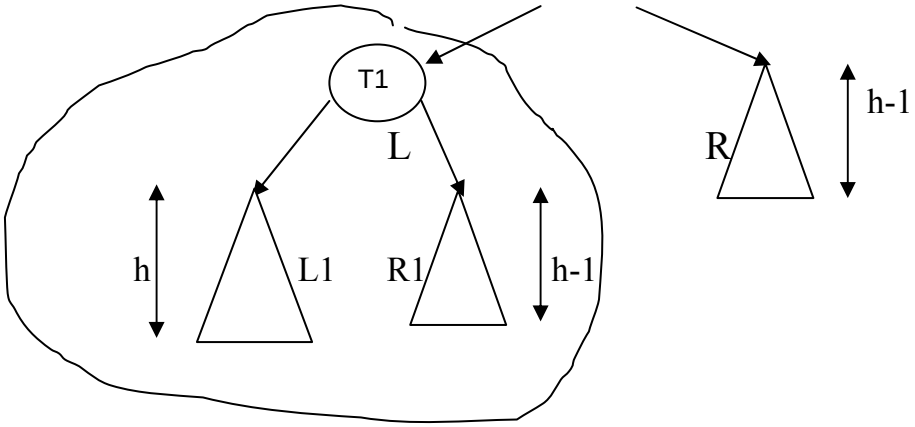
Việc *thêm hay hủy một nút trên cây AVL* có thể làm cây tăng hay giảm chiều cao, khi đó ta cần phải cân bằng lại cây. Để *giảm tối đa chi phí cân bằng lại cây*, ta *chỉ cân bằng lại cây AVL ở phạm vi cục bộ*.

Các trường hợp mất cân bằng

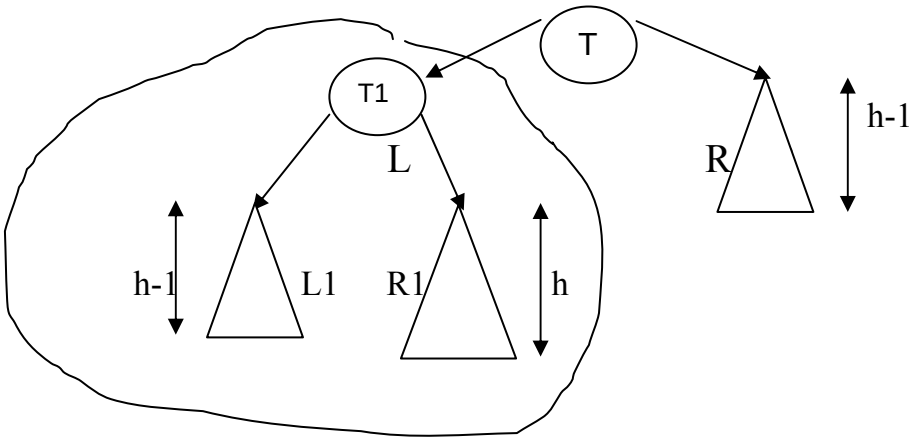
Ngoài các thao tác thêm và hủy, đối với cây cân bằng, ta còn có thêm thao tác cơ bản là *cân bằng lại* cây AVL trong trường hợp thêm hoặc hủy một nút của nó. Khi đó, độ lệch giữa chiều cao cây con phải và trái sẽ là 2. Do các trường hợp cây lệch trái và phải tương ứng là đối xứng nhau, nên ta chỉ xét trường hợp cây AVL lệch trái.

<u>*Trường hợp a*</u>: cây con T1 lệch trái

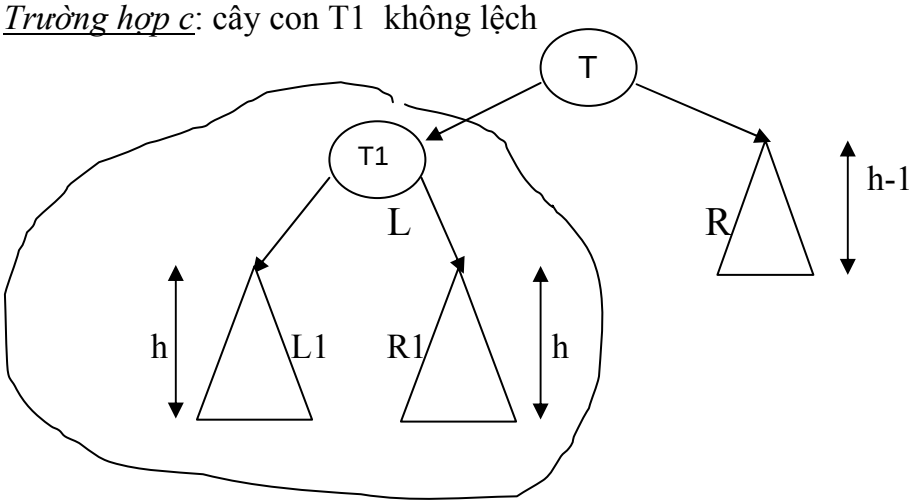




Trường hợp b: cây con T1 lệch phải

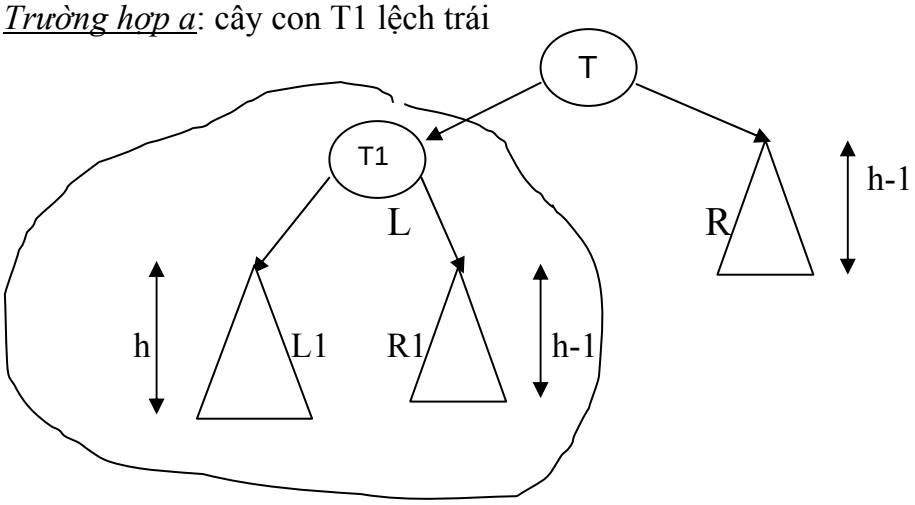


Trường hợp c: cây con T1 không lệch

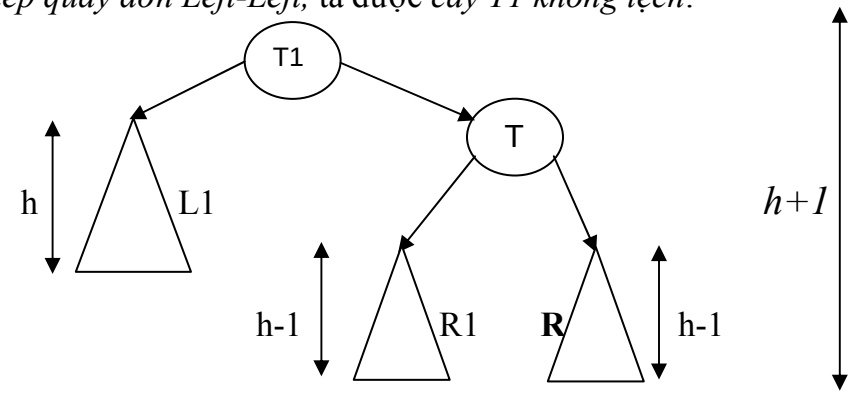


Việc *cân bằng lại* trong trường hợp *b* (cây con T1 lệch phải) là phức tạp nhất.

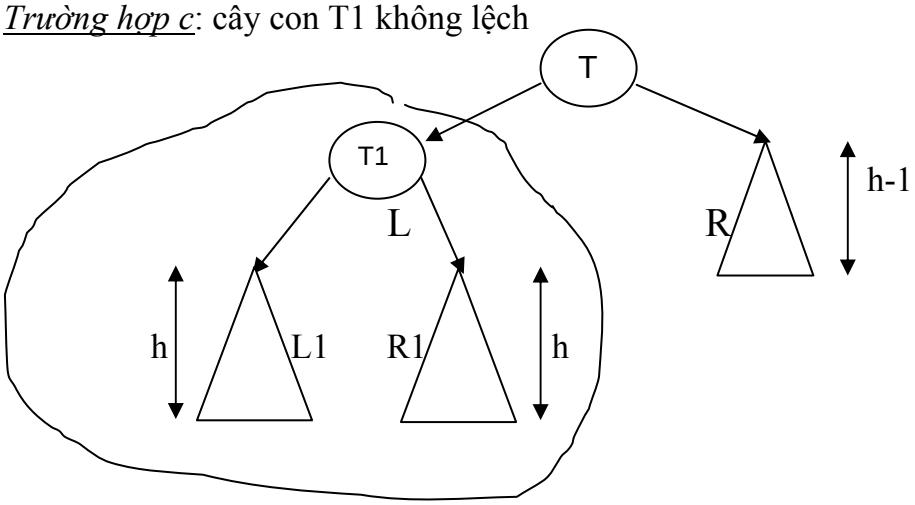
Trường hợp a: cây con T1 lệch trái



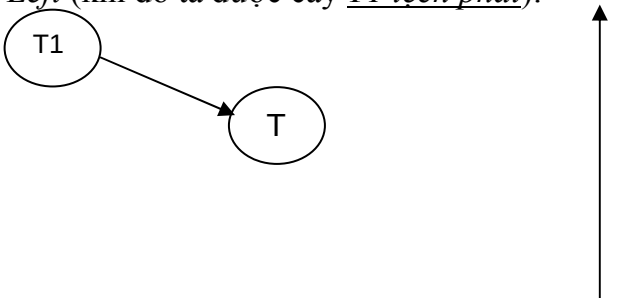
Cân bằng lại bằng *phép quay đơn Left-Left,* ta được *cây T1 không lệch*:

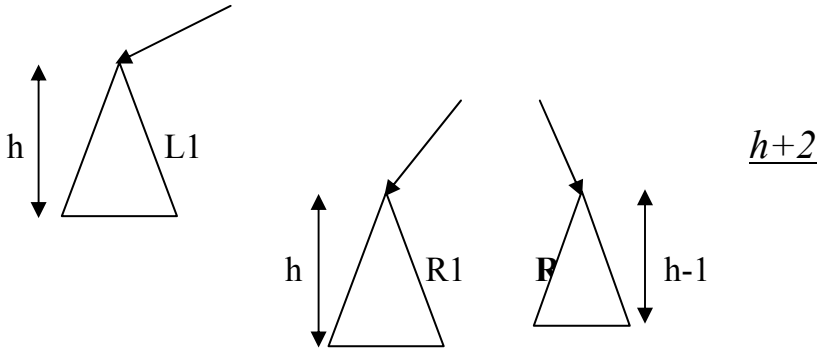


Trường hợp c: cây con T1 không lệch

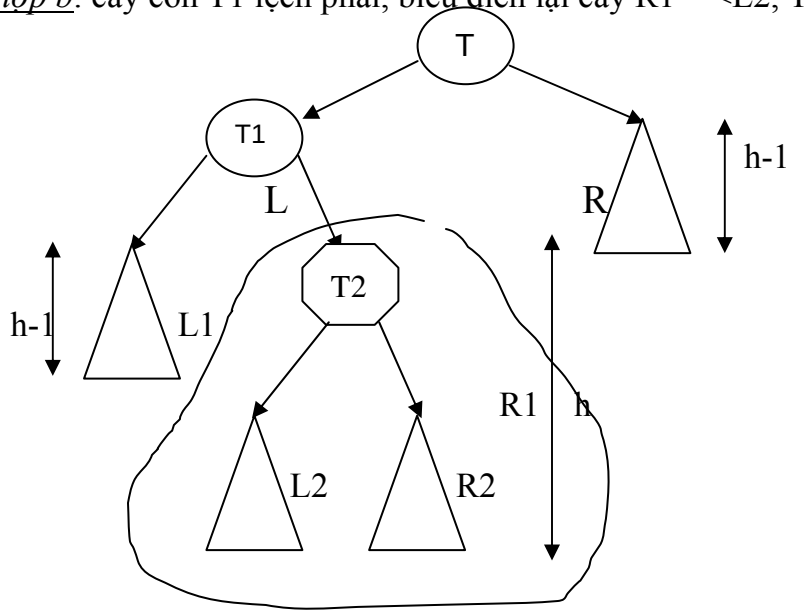


Cân bằng lại bằng *phép quay đơn Left-Left* (khi đó ta được cây *T1 lệch phải*):

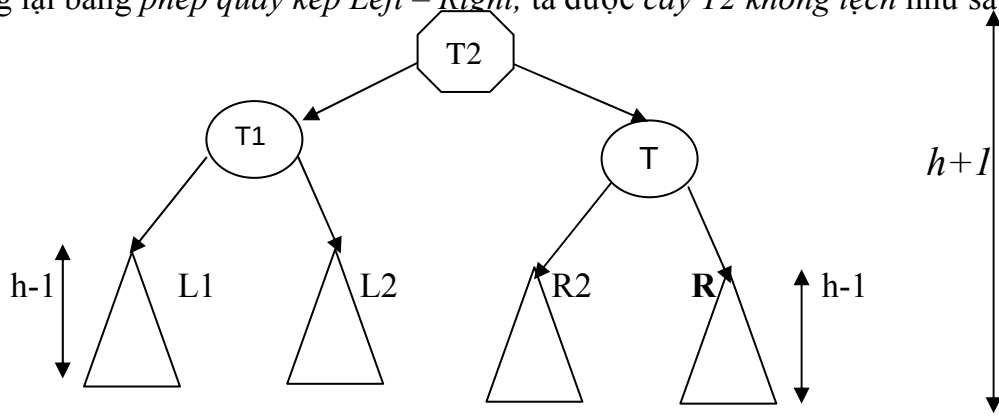




Trường hợp b: cây con T1 lệch phải, biểu diễn lại cây R1 = <L2, T2, R2> như sau:



Cân bằng lại bằng *phép quay kép Left – Right,* ta được *cây T2 không lệch* như sau:



* *Nhận xét*:

- *Trước khi cân bằng lại,* cây *T lệch (và mất cân bằng) và* có chiều cao là *h+2* trong cả 3 trường hợp. Nhưng *sau khi cân bằng lại* cây *T,* nó *vẫn lệch (lệch phải, nhưng tất nhiên vẫn cân bằng)* và có chiều cao là *h+2* chỉ trong trường hợp *c*; còn trong hai trường hợp *a* và *b*, cây *T mới (là*

T1 hay T2 tương ứng với trường hợp a hay b) không lệch và có chiều cao là *h+1*.

- *Các thao tác cân bằng lại trong mọi trường hợp đều có độ phức tạp là O(1).*

Sau đây là phần cài đặt các phép quay đơn và kép cho cây T mất cân bằng trong hai trường hợp nó bị lệch trái và lệch phải.

```
//Phép quay đơn Left – Left
void RotateLL(AVLTree &T)
{ AVLTree T1 = T->Lchild;
        T->Lchild = T1->Rchild;
        T1->Rchild = T;
        switch (T1->Balfactor)
        {case LH: T->Balfactor = EH;
                  T1->Balfactor = EH; break;
          case EH: T->Balfactor = LH;
                  T1->Balfactor = RH; break;
        }
        T = T1;
        return ;
}

//Phép quay đơn Right – Right
void RotateRR (AVLTree &T)
{ AVLTree T1 = T->Rchild;
        T->Rchild = T1->Lchild;
        T1->Lchild = T;
        switch (T1->Balfactor)
        {case RH: T->Balfactor = EH;
                  T1->Balfactor = EH; break;
          case EH: T->Balfactor = RH;
                  T1->Balfactor = LH; break;
        }
        T = T1;
        return ;
}

//Phép quay kép Left – Right
void RotateLR(AVLTree &T)
{ AVLTree T1 = T->Lchild, T2 = T1->Rchild;
        T->Lchild = T2->Rchild; T2->Rchild = T;
        T1->Rchild = T2->Lchild; T2->Lchild = T1;
```

```
        switch (T2->Balfactor)
        {case LH: T->Balfactor = RH;
                  T1->Balfactor = EH; break;
         case EH: T->Balfactor = EH;
                  T1->Balfactor = EH; break;
         case RH: T->Balfactor = EH;
                  T1->Balfactor = LH; break;
        }
        T2->Balfactor = EH;
        T = T2;
        return ;
}

//Phép quay kép Right-Left
void RotateRL(AVLTree &T)
{ AVLTree T1 = T->RLchild, T2 = T1->Lchild;
        T->Rchild = T2->Lchild; T2->Lchild = T;
        T1->Lchild = T2->Rchild; T2->Rchild = T1;
        switch (T2->Balfactor)
        {case LH: T->Balfactor = EH;
                  T1->Balfactor = RH; break;
         case EH: T->Balfactor = EH;
                  T1->Balfactor = EH; break;
         case RH: T->Balfactor = LH;
                  T1->Balfactor = EH; break;
        }
        T2->Balfactor = EH;
        T = T2;
        return ;
}
```

Sau đây là thao tác ***cân bằng lại*** khi cây bị *lệch trái* hay *lệch phải*.

```
//Cân bằng lại khi cây bị lệch trái
int LeftBalance(AVLTree &T)
{ AVLTree T1 = T->Lchild;
        switch (T1->Balfactor)
        { case LH : RotateLL(T); return 2; //cây T giảm độ cao và không bị lệch
          case EH : RotateLL(T); return 1;//cây T không giảm độ cao và bị lệch phải
          case RH : RotateLR(T); return 2;
        }
        return  0;
}
```

```
//Cân bằng lại khi cây bị lệch phải
int RightBalance(AVLTree &T)
{ AVLTree T1 = T->Rchild;
        switch (T1->Balfactor)
        { case LH : RotateRL(T); return 2; //cây T không bị lệch
          case EH : RotateRR(T); return 1; //cây T bị lệch trái
          case RH : RotateRR(T); return 2;
        }
        return  0;
}
```

IV.4.4. Chèn một phần tử vào cây AVL

Việc chèn một phần tử vào cây AVL xảy ra tương tự như trên cây nhị phân tìm kiếm. Tuy nhiên, sau khi chèn xong, nếu chiều cao của cây thay đổi tại vị trí thêm vào, ta phải lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng hay không. Nếu có, ta *chỉ phải cân bằng lại ở nút này*. (Việc cân bằng lại chỉ cần thực hiện một lần tại nơi mất cân bằng)

Hàm chèn trả về các trị –1, 0, 1 hay 2 tương ứng khi: không đủ bộ nhớ cấp phát cho một nút của cây hoặc gặp nút đã có trên cây hoặc thành công hoặc chiều cao của cây bị tăng sau khi chèn.

Khi chèn một nút vào cây AVL, ta cần sử dụng hàm cấp phát bộ nhớ cho một nút của cây AVL.

```
AVLTree CấpPhátAVL()
{ AVLTree Tam= new AVLTreeNode;
        if (Tam == NULL)
                cout << "\nKhông đủ bộ nhớ cấp phát cho một nút của cây AVL !";
        return Tam;
}

int  ChènAVL( AVLTree &T, ElementType x)
{ int  Kquả;
        if (T)
                { if  (T->Data == x) return 0; //Đã có nút trên cây
                  if  (T-> Data >  x)
                        { Kqủa=ChènAVL(T->Lchild,x);//chèn x vào cây con trái của T
                          if (Kqủa < 2) return  Kqủa;
                          switch  (T->Balfactor)
                          { case LH: LeftBalance(T); return 1;//trước khi chèn,T lệch trái
                            case EH: T->Balfactor=LH;return 2;//trước khi chèn,T không lệch
```

```
                    caseRH:T->Balfactor=EH; return 1;//trước khi chèn,T lệch phải
                  }
                }
            else  // T-> Data <  x
                { Kqủa=ChènAVL(T->Rchild,x);//chèn x vào con phải của T
                  if (Kqủa < 2) return  Kqủa;
                  switch  (T->Balfactor)
                  { case LH: T->Balfactor = EH; return 1; //trước khi chèn,T lệch trái
                    case EH:T->Balfactor=RH;return 2;//trước khi chèn,T không lệch
                    case RH : RightBalance(T); return 1; //trước khi chèn,T lệch phải
                  }
                }
          }
    else //T==NULL
          { if ((T = CấpPhátAVL()) == NULL) return –1; //Thiếu bộ nhớ
            T->Data = x;
            T->Balfactor = EH;
            T->Lchild  = T->Rchild = NULL;
            return 2; //thành công và chiều cao của cây tăng
          }
}
```

IV.4.5. Xóa một phần tử khỏi cây AVL

Việc xóa một phần tử ra khỏi cây AVL diễn ra tương tự như đối với cây nhị phân tìm kiếm; chỉ khác là sau khi hủy, nếu cây AVL bị mất cân bằng, ta phải cân bằng lại cây. Việc *cân bằng lại cây có thể xảy ra phản ứng dây chuyền*.

Hàm XóaAVL sẽ trả về trị 1 hoặc 0 hoặc 2 tùy theo việc hủy thành công hoặc không có x trên cây hoặc sau khi hủy, chiều cao của cây bị giảm.

int XóaAVL(AVLTree &T, ElementType x)

```
{ int Kqủa;
    if (T== NULL) return 0; // không có x trên cây
    if  (T-> Data >  x)
          { Kqủa = XoáAVL(T->Lchild,x); // tìm và xóa x trên cây con trái của
T
            if (Kqủa < 2) return  Kqủa;
            switch  (T->Balfactor)
            { case LH : T->Balfactor = EH; return 2; //trước khi xóa,T lệch trái
              case EH : T->Balfactor = RH; return 1;//trước khi xóa,T không lệch
              case RH : return RightBalance(T);      //trước khi xóa,T lệch phải
            }
          }
```

```
        else if (T-> Data <  x)
                { Kqủa = XoáAVL(T->Rchild,x); // tìm và xóa x trên cây con phải của T
                  if (Kqủa < 2) return  Kqủa;
                  switch  (T->Balfactor)
                  { case LH : return LeftBalance(T);        //trước khi xóa,T lệch trái
                    case EH : T->Balfactor = LH; return 1;//trước khi xóa,T không lệch
                    case RH : T->Balfactor = EH; return 2; //trước khi xóa,T lệch phải
                  }
                }
              else //T->Data== x
                 { AVLTree p = T;
                   if (T->Lchild == NULL)
                         { T = T->Rchild; Kqủa = 2;
                         }
                   else  if  (T->Rchild == NULL)
                                { T = T->Lchild; Kqủa = 2;
                                }
                         else // T có cả 2 con
                                { Kqủa = TìmPhầnTửThayThế(p,T->Rchild);
                                                // tìm phần tử thay p để xóa trên nhánh phải của
                                                T
                                  if (Kqủa == 2) switch (T->Balfactor)
                                        { case  LH : Kquả = LeftBalnce(T); break;
                                          case  EH: T->Balfactor=LH; Kquả = 1; break;
                                          case  RH: T->Balfactor=EH; Kquả = 2; break;
                                        }
                                }
                   delete p;
                   return Kquả;
                 }
}

// Tìm phần tử  thay thế
int TìmPhầnTửThayThế(AVLTree  &p, AVLTree  &q)
{ int Kqủa;
        if (q->Lchild)
                { Kqủa = TìmPhầnTửThayThế(p, q->Lchild);
                  if (Kqủa < 2) return Kquả;
                  switch (q->Balfactor)
                  { case  LH : q->Balfactor = EH; return 2;
                    case  EH : q->Balfactor = RH; return 1;
                    case  RH : return RightBalance(q);
                  }
```

```
            }
        else    { p->Data = q->Data;
                  p = q;
                  q = q->Rchild;
                  return 2;
                }
}
```

* *Nhận xét*:

- Thao tác thêm một nút có độ phức tạp O(1).
- Thao tác huỷ một nút có độ phức tạp O(h)
- Với cây cân bằng, trung bình: 2 lần thêm vào cây thì cần 1 lần cân bằng lại, 5 lần hủy thì cần 1 lần cân bằng lại.
- Việc hủy một nút có thể phải cân bằng dây chuyền các nút từ gốc đến phần tử bị hủy, trong khi thêm vào 1 nút chỉ cần 1 lần cân bằng cục bộ.
- Độ dài đường tìm kiếm trung bình trong cây AVL gần bằng cây cân bằng hoàn toàn ($log_2 n$), nhưng việc cân bằng lại đơn giản hơn nhiều.
- Một cây cân bằng AVL không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng.

BÀI TẬP
"CẤU TRÚC DỮ LIỆU & GIẢI THUẬT 1"

Mục đích các bài tập:

- Kiểm tra, củng cố việc hiểu các cấu trúc dữ liệu và các thuật toán có liên quan.
- Rèn luyện kỹ năng lập trình và vận dụng lý thuyết vào việc chọn lựa các cấu trúc dữ liệu và các thuật toán phù hợp có liên quan cho một bài toán cụ thể.
- Phát triển và tổng hợp các kết quả lý thuyết nhằm chuẩn bị cho học viên làm quen với quá trình giải quyết hoàn chỉnh một bài toán không tầm thường nào đó.

Các bài tập có đánh dấu (*) là các bài tập khó hoặc cần nhiều thời gian để thực hiện dành cho các học viên khá giỏi. Có thể kết hợp nhiều bài tập (*) có liên quan hoặc bổ sung thêm các ứng dụng thực tế để hình thành tiểu luận của môn học. Phần in đậm có gạch chân là yêu cầu tối thiểu học viên cần thực hiện trong giờ thực hành.

Bài tập chương I (Giới thiệu cấu trúc dữ liệu, phân tích thuật toán)

(Kiểu dữ liệu có cấu trúc)

1) Giả sử quy tắc tổ chức quản lý nhân viên của một công ty như sau:

- Thông tin về một nhân viên bao gồm lý lịch và bảng chấm công:

* *Lý lịch nhân viên*:

- Mã nhân viên : chuỗi 10 ký tự
- Tên nhân viên : chuỗi 30 ký tự
- Tình trạng gia đình : 1 ký tự (M = Married, S = Single)
- Số con : số nguyên ≤ 20
- Trình độ văn hoá : chuỗi 2 ký tự (C1 = cấp 1,C2=cấp 2,C3=cấp 3; DH = đại học, CH = cao học, TS = tiến sĩ)
- Lương căn bản : số ≤ 1 000 000

* *Chấm công nhân viên*:

- Số ngày nghỉ có phép trong tháng : số ≤ 28
- Số ngày nghỉ không phép trong tháng : số ≤ 28
- Số ngày làm thêm trong tháng : số ≤ 28
- Kết quả công việc : chuỗi 2 ký tự (T = tốt, TB = trung bình, K = Kém)
- Lương thực lĩnh trong tháng : số ≤ 2 000 000

- Quy tắc tính lương:
  Lương thực lĩnh = Lương căn bản + Phụ trội
  Trong đó nếu:
  - số con > 2 : Phụ trội = +5% Lương căn bản
  - trình độ văn hoá = CH : Phụ trội = +10% Lương căn bản
  - làm thêm : Phụ trội = +4% Lương căn bản / 1 ngày
  - nghỉ không phép : Phụ trội = -5% Lương căn bản / 1 ngày

- Các chức năng yêu cầu:
  - Cập nhật lý lịch, bảng chấm công cho nhân viên (thêm, xóa, sửa một hay mọi mẫu tin thoả mãn một tính chất nào đó)
  - Xem bảng lương hàng tháng
  - Khai thác (chẳng hạn tìm) thông tin của nhân viên

  Hãy chọn cấu trúc dữ liệu thích hợp (và giải thích tại sao?) để biểu diễn các thông tin trên và cài đặt chương trình theo các chức năng đã mô tả. Biết rằng số nhân viên tối đa là 50 người, chú ý các thông tin tĩnh và "động" hay thay đổi và là hệ quả của những thông tin khác.

2) Viết chương trình cài đặt chuỗi ký tự theo một trong hai cách (giả sử kiểu chuỗi chưa có sẵn trong ngôn ngữ lập trình bạn đang dùng) sau:

a. phần tử đầu chỉ số ký tự của chuỗi;

b. chuỗi được kết thúc bởi ký tự có mã ASCII bằng 0.

Sau đó viết lại các thao tác cơ bản trên chuỗi (tính chiều dài chuỗi, nối, sao chép một phần của chuỗi, chặt ngắn chuỗi, kiểm tra chuỗi con, ...)

(Độ phức tạp của thuật toán)

3) Hãy nêu một thuật toán mà độ phức tạp tính toán của nó là: O(1), O(n), $O(n^2)$.

4) Hãy xác định mục đích của từng thuật toán sau (xác định phép toán đặc trưng cơ bản của nó) và tính độ phức tạp tính toán của nó trong trường hợp xấu nhất, tốt nhất:

a)
```
Sum = 0;
for  (i = 1; i <= n; i++)
{ cin  >> x;                  // Nhập một số x;
  Sum = Sum + x;
}
```

b)
```
for  (i = 1; i <= n; i++)
      for  ( j = 1; j <= n; j++)
      { C[i,j] = 0;
        for  (k = 1; k <= n; k++) C[i,j] = C[i,j] + A[i,k]*B[k,j];
      }
```

c)
```
for  (i = 1; i <= n -1; i++)
{ for  ( j = i; j <= n -1; j++)
      if  (X[ j] > X[ j+1])
```

```
            { Temp = X[ j];
              X[ j] = X[ j+1];
              X[ j+1] = Temp;
            };
        }
```

d) (*) int Max(int i, int n) // x là mảng các số nguyên; $n=2^k>=i$; gọi Max(1, n)

```
        { int m1, m2;
            if (n == i) return x[n-1];
            else { m1 = Max(i, (n+i)/2);
                   m2 = Max((n+i)/2+1, n);
                   if (m1 < m2) return m2;
                   else return m1;
                 }
        }
```

5) Viết giải thuật đệ qui và giải thuật lặp để:

a) Tính ước số chung lớn nhất của 2 số nguyên không âm.

b) Tính tổ hợp chập k của n phần tử

c) Tìm chuỗi đảo ngược của một chuỗi ký tự cho trước.

Bài tập chương II (Tìm kiếm và sắp xếp trên mảng)

(Tìm kiếm)

1) Xét các dãy số nguyên sau:

α.	-9	-9	-5	-2	0	3	7	7	10	15
β.	15	10	7	7	3	0	-2	-5	-9	-9
γ.	66,	22,	36,	6,	79,	26,	45,	75,	13,	31,
	62,	27,	76,	33,	16,	47				

Với mỗi mảng số nguyên, hãy:

a. Đếm số lần tìm kiếm (so sánh) trung bình một phần tử *x* nào đó trên dãy (*x* có thể có hoặc không có mặt trong dãy);

b. Kiểm tra lại kết quả câu a) bằng một chương trình trên máy tính và so sánh lại với kết quả đánh giá độ phức tạp của các thuật toán:

- tìm kiếm tuyến tính (trên dãy chưa được hoặc đã được sắp tăng),
- tìm kiếm nhị phân.

2) Xây dựng và cài đặt thuật toán tìm:

a. phần tử lớn nhất (hay nhỏ nhất),

b. tất cả các số nguyên tố,

c. tìm phần tử đầu tiên trên dãy mà thỏa một tính chất TC nào đó;

d. (*) dãy con (là một dãy các phần tử liên tiếp của dãy) tăng dài nhất, trong một dãy các phần tử cho trước được cài đặt bằng mảng.

3) (*) Xây dựng và cài đặt thuật toán tìm phần tử median (phần tử đứng giữa về mặt giá trị) trong một dãy được cài đặt bằng mảng.

(Sắp xếp)

4) Với mỗi bộ dữ liệu của bài tập 1), hãy:

a. Thực hiện từng bước và đếm số phép so sánh và gán trong các thuật toán sắp xếp tăng dãy đã cho;

b. Kiểm tra lại kết quả ở câu a) bằng một chương trình trên máy tính;

c. (*) Tổng quát câu b) trên bộ dữ liệu lớn được tạo ra tự động một cách ngẫu nhiên trong ba tình huống: xấu nhất, tốt nhất và trung bình ngẫu nhiên; thống kê các kết quả trên và thời gian chạy của từng thuật toán dưới dạng bảng;

d. **(**)** Thể hiện trực quan bằng đồ thị kết quả của câu c) và cho nhận xét bằng các phương pháp sắp xếp sau:

- sắp đổi chỗ trực tiếp BubbleSort, ShakerSort và QuickSort,
- sắp chèn trực tiếp và ShellSort,
- sắp chọn trực tiếp và HeapSort,
- sắp trộn tự nhiên,
- sắp dựa trên cơ số RadixSort.

5) Hãy viết thuật toán và chương trình sắp xếp bằng phương pháp *chọn hai đầu*: tại mỗi bước chọn đồng thời cả phần tử nhỏ nhất và lớn nhất trong dãy chưa được sắp còn lại.

6) (*) Cho các ví dụ để minh họa ưu điểm của các thuật toán sắp xếp cải tiến so với các thuật toán sắp xếp trực tiếp tương ứng.

7) Xét thuật toán phân hoạch trong thuật toán QuickSort được viết lại như sau:

```
i = 0; j = n -1; y = x[n/2];
do
{ while (x[i] < y) i++;
  while (x[ j] > y) j--;
  HoánVị(x[i], x[ j]);
} while (i <= j);
```

Có bộ dữ liệu x[0], x[1], …, x[n-1] nào làm đoạn chương trình trên sai hay không ? Cho ví dụ minh họa.

8) Viết hàm đếm số đường chạy (tự nhiên) của một dãy gồm *n* phần tử cho trước.

9) Hãy cài đặt thêm thuật toán xuất bảng lương nhân viên (trong bài tập 1 - chương 1) theo thứ tự tiền lương tăng dần.

10) (*) Hãy viết lại giải thuật QuickSort dưới dạng lặp.

11) (*) Cải tiến hai thuật toán QuickSort viết dưới dạng đệ qui và lặp [*gợi ý*: ta nên thực hiện sắp xếp trước dãy con nào ngắn hơn].

12) (*) Xây dựng ví dụ để trường hợp xấu nhất của thuật toán QuickSort xảy ra.

Bài tập chương III (Cấu trúc danh sách liên kết)

1) Xét đoạn chương trình tạo một DSLK đơn có 4 nút (không quan tâm đến dữ liệu) sau đây:

```
NodePointer p, Dx = NULL;
p = Dx; Dx = new NodeType;
for (i = 0; i < 4; i++)
{ p = p->Next;
  p = new NodeType;
}
p->Next = NULL;
```

Đoạn chương trình này có thực hiện đúng như mục đích đã đưa ra không ? Tại sao ? Nếu không thì cần sửa lại như thế nào cho đúng ?

2) Hãy thực hiện các yêu cầu sau đối với từng loại danh sách liên kết:

i) DSLK không có nút câm
ii) DSLK có nút câm
iii) DSLK vòng (không có nút câm)
iv) DSLK đối xứng
v) DSLK vòng đôi

a. Tạo bản sao của một DSLK cho trước.
b. Nối hai DSLK cho trước.
c. Tính số lượng các nút dữ liệu.
d. Tìm nút dữ liệu đầu tiên trong DSLK thỏa một tính chất nào đó, chẳng hạn:
   - nút thứ k,
   - hoặc có trường dữ liệu trùng với một giá trị cùng kiểu K cho trước.

   Nếu có thì trả về con trỏ chỉ đến nút đứng trước nút tìm thấy.
e. Xóa một (hay mọi) nút dữ liệu trong DSLK thỏa một tính chất nào đó, ví dụ:
   - nút thứ k,
   - hoặc có trường dữ liệu trùng với một giá trị cùng kiểu K cho trước.
f. Bổ sung một nút L vào sau một (hay mọi) nút dữ liệu trong DSLK thỏa một tính chất nào đó, chẳng hạn:
   - nút thứ k,
   - hoặc có trường dữ liệu trùng với một giá trị cùng kiểu K cho trước.
g. Đảo ngược DSLK nói trên theo hai cách : tạo DSLK mới hay sửa lại chiều con trỏ trong DSLK ban đầu.
h. Gọi M là con trỏ chỉ tới một nút đã có trong DSLK trên và P là con trỏ chỉ tới một DSLK khác cùng loại. Hãy chèn DSLK P này vào sau nút trỏ bởi M.
i. Tách thành 2 DSLK mà DS sau được trỏ bởi M (giả thiết như câu h).
j. So sánh 2 DSLK (có trường dữ liệu của các nút liên tiếp tương ứng bằng nhau hay không ?)

3) Hãy viết chương trình nhằm thực hiện các yêu cầu của bài tập 1 – chương 1 (biết rằng số lượng nhân viên biến động nhiều, không dự đoán được giới hạn của nó) bằng cách dùng DSLK để cài đặt.

4) Hãy viết thuật toán và chương trình để trộn hai DSLK tăng A, B cho trước thành một DSLK C cũng tăng theo hai cách:

a. C là DSLK mới (cấp phát bộ nhớ mới cho mọi nút của C) và bảo toàn hai DSLK cũ A, B;

b. C là DSLK mới do A, B hợp thành (do đổi chỗ vị trí các con trỏ sẵn có trên A, B). Khi đó cấu trúc hai DSLK A, B có thể bị thay đổi.

5) Một số giới hạn vé (MAX_VE) cho buổi hòa nhạc sẽ được bán vào ngày mai. Người nào đăng ký trước sẽ được mua trước. Hãy viết một chương trình:

a. Đọc các tên, tuổi của những người đăng ký cùng với số vé họ mua và lưu vào một DSLK (chú ý kiểm tra không có người nào được đăng ký nhiều lần).

b. Hiện ra màn hình DSLK trên.

6) *(Bài toán Josephus)* Một nhóm binh sĩ bị kẻ thù bao vây và một binh sĩ được chọn để đi cầu cứu. Việc chọn được thực hiện theo cách sau đây. Một số nguyên n và một binh sĩ được chọn ngẫu nhiên. Các binh sĩ được sắp theo vòng tròn và họ đếm từ binh sĩ được chọn ngẫu nhiên. Khi đạt đến n, binh sĩ tương ứng được lấy ra khỏi vòng và việc đếm lại được bắt đầu từ binh sĩ tiếp theo. Quá trình này tiếp tục cho đến khi chỉ còn lại một binh sĩ là người gặp may (hoặc không may) được chọn để đi cầu cứu. Hãy viết một thuật toán cài đặt cách chọn này, dùng danh sách liên kết vòng để lưu trữ các tên của binh sĩ.

(Ngăn xếp và hàng đợi)

7) Cho X là ngăn xếp chứa các ký tự. Giả sử có hàm sau trong C++:

```
void Out(StackType &S, ElementType  &Item)
{       Pop(S,Item);  cout << Item<< endl;
}
```

Ta cần sử dụng luân phiên các phép toán Push(S, Item) và Out(S,Item) như thế nào (nếu có thể) từ bộ các ký tự : 'A', 'B', 'C', 'D', 'E', 'F' để thu được các anagram (hoán vị) sau đây của nó:

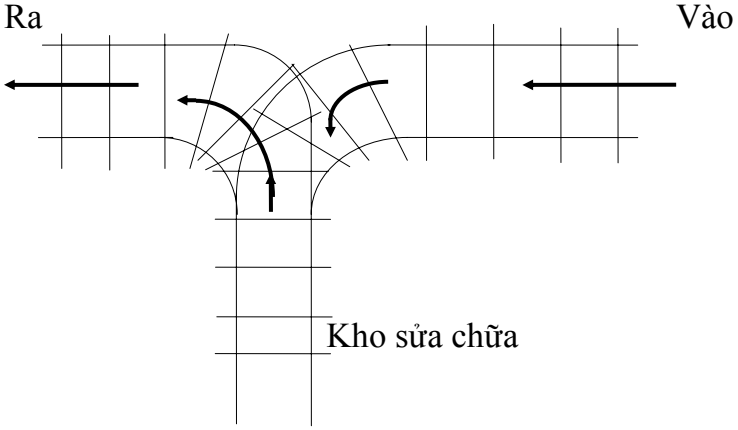
a) BDCFEA
b) BDACEF
c) ABCDEF
d) EBFCDA
e) FEDCBA

8) Xét một cơ cấu đường tàu và kho sửa chữa như hình sau:

Giả sử ở đường vào có 4 đường tàu được đánh số 1, 2, 3, 4. Gọi V là phép đưa một đầu tàu vào kho sửa chữa, R là phép đưa một đầu tàu ra khỏi kho.

a. Nếu thực hiện dãy VVRVVRRR thì thứ tự các đầu tàu lúc ra là gì ? (Có thể xem đây là một cách hoán vị các số được không ?)

b. Xét trường hợp có 6 đầu tàu:1, 2, 3, 4, 5, 6 có thể thực hiện một dãy các phép V và R thế nào để đổi thứ tự đầu tàu ở đường ra là: 3, 2, 5, 6, 4, 1 ? và 1, 5, 4, 6, 2, 3 ?



9) Xét chuỗi:

EAS*Y**QUE***ST***I*ON

Trong đó, mỗi chữ cái tượng trưng cho thao tác thêm nó vào một DSLK List, dấu * tượng trưng cho thao tác lấy nó ra khỏi List và xuất ra màn hình.

Trong từng trường hợp sau, với List là:

a. ngăn xếp

b. hàng đợi

hãy cho biết:

- Nội dung của List sau mỗi thao tác cơ bản trên ?
- Kết quả cuối cùng xuất ra trên màn hình ?
- Hãy kiểm tra lại các kết quả trên bằng một chương trình hoàn chỉnh.

10) Viết các thao tác cơ bản trên ngăn xếp và thêm vào các thao tác sau đây:

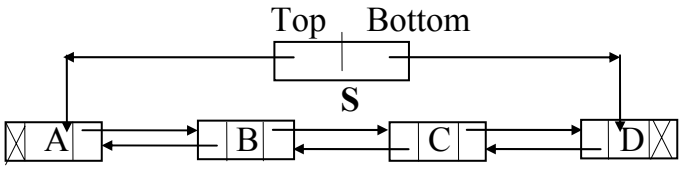
a. ElementType *XemPTửThứ_2CủaNX*(StackType S) có tác dụng xem phần tử thứ 2 kể từ đỉnh ngăn xếp S mà không làm S thay đổi.

b. ElementType *LấyPTửThứ_2CủaNX*(StackType &S) có tác dụng trả về phần tử thứ 2 của ngăn xếp S và S bị mất đi 2 phần tử ở đỉnh của nó.

c. ElementType *LấyĐáyNX*(StackType &S) có tác dụng trả về phần tử ở đáy ngăn xếp S và làm S trở thành rỗng.

d. ElementType *XemĐáyNX*(StackType S) có tác dụng trả về phần tử ở đáy ngăn xếp S và S không thay đổi.

11) Để có thể duyệt ngăn xếp hay hàng đợi theo cả hai chiều, ta có thể tổ chức chúng theo kiểu DSLK đối xứng như sau:



Hãy thực hiện các phép toán sau trên ngăn xếp:

a. Thực hiện phép duyệt qua DSLK từ dưới lên.

b. Thực hiện phép duyệt qua DSLK từ trên xuống.

c. Thực hiện phép bổ sung một phần tử vào (đầu và đuôi) DSLK.

d. Thực hiện phép loại bỏ một phần tử (ở đầu và đuôi) khỏi DSLK.

12) a. Cho Q là hàng đợi rỗng. Cho biết kết quả của Q sau một dãy các phép toán thêm vào và lấy ra các ký tự sau đây:
EnQueue(Q, 'A'), EnQueue(Q, 'B'), EnQueue(Q, 'C'), DeQueue(Q, Item),
EnQueue(Q, 'D'), EnQueue(Q, 'E'), DeQueue(Q, Item), DeQueue(Q, Item),
EnQueue(Q, 'F'), DeQueue(Q, Item).

b. Viết các thao tác cơ bản trên hàng đợi và thêm vào các thao tác sau đây: duyệt hàng đợi từ đầu đến đuôi của nó và ngược lại.

13) Dùng các phép toán cơ bản trên ngăn xếp và hàng đợi để đảo ngược thứ tự các phần tử trên hàng đợi.

14) Phân tích một số thành tích các thừa số nguyên tố theo thứ tự giảm dần. Ví dụ: phân tích: 60 = 5*3*2*2.

15) Dùng ngăn xếp để kiểm tra một chuỗi ký tự S1 có phải là palyndrome của một chuỗi ký tự S2 hay không ?

16) (*) Viết một chương trình đọc một xâu ký tự chứa các dấu ngoặc và xác định xâu đó có chứa các dấu ngoặc tương ứng hợp lệ hay không. Ví dụ:

- các xâu sau là hợp lệ: a*(b+c), a(), b[d(e+f-)], d-{[a(b)d]}
- các xâu sau là không hợp lệ: (,], a*(b+c], a[), b[d(e+f-]), d-{[a((b)d]}

(Các ứng dụng khác của DSLK)

17) **a.** Chuyển các biểu thức trung tố sau đây sang dạng hậu tố:
a/(b*c), a/b*c, a∧b∧c, (a∧b)∧c, a-b-c, a-(b-c), $a^5 + 4a^3 - 3a^2 + 7$, (a+b)*(c-d), S^{a+b}

b. Viết biểu thức sau đây dưới dạng hậu tố: (A * B)/(C + D). Minh họa thông qua hình ảnh Stack để tính giá trị biểu thức hậu tố này ứng với: A= 20, B = 4, C = 9, D = 7.

c. **(**)** Cài đặt một chương trình để :

i) Chuyển một biểu thức từ dạng trung tố sang dạng hậu tố (có kiểm tra cú pháp của biểu thức).

ii) Tính giá trị của một biểu thức cho trước ở dạng hậu tố.

iii) Vẽ đồ thị của một hàm giải tích cho trước được đưa vào dưới dạng biểu thức chuỗi.

iv) Có thể viết lại chương trình trên khái quát hơn để có thể áp dụng cho các biểu thức lôgic mệnh đề hay không ?

18) (**) Hãy viết một chương trình thực hiện các yêu cầu tương tự của bài tập 4 - chương 2 để cài đặt các thuật toán sắp xếp sau trên DSLK động (DSLK đơn, DSLK kép):

a. QuickSort

b. MergeSort

c. RadixSort

d. Các phương pháp sắp xếp trực tiếp: chèn, chọn, đổi chỗ

19) (*) Hãy lập các giải thuật cộng, trừ, nhân hai đa thức và tính đạo hàm, nguyên hàm của một đa thức cho trước trong hai trường hợp:

a) Khi các hệ số của đa thức được lưu đầy đủ trong mảng.

b) (*) Khi chỉ các hệ số khác không và các số mũ tương ứng được lưu trong một danh sách liên kết.

20) (*) Hãy cài đặt tập hợp bằng DSLK và thực hiện các phép toán trên tập hợp (quan hệ một phần tử có thuộc vào một tập không; quan hệ bao hàm, bằng nhau giữa hai tập; phép toán giao, hiệu, hợp hai tập hợp, ...)

21) (**) Viết các phép toán cơ bản trên ma trận thưa được cài đặt bằng DSLK tổng quát.

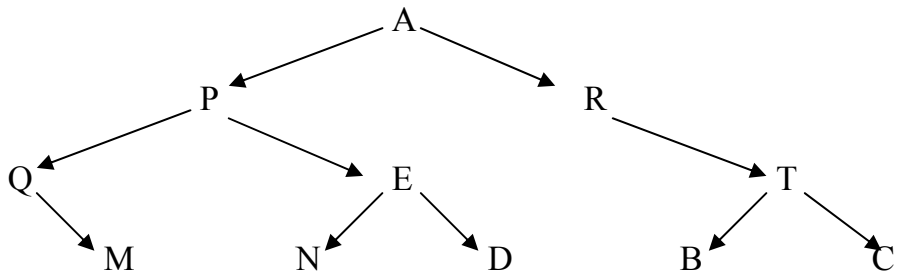
22) a. Hãy cài đặt các thao tác cơ bản trên DSLK có thứ tự và tổ chức lại, hàng đợi ưu tiên. So sánh thời gian tìm kiếm của cách tổ chức này với các cách tổ chức bình thường.

b. Tìm một ứng dụng thực tế của hàng đợi ưu tiên.

23) (*) Áp dụng thuật toán sắp xếp tôpô vào bài toán sắp lịch giảng dạy (tuyến tính) cho dãy các học phần thỏa điều kiện "học trước" đã biết.

Bài tập chương IV (Cấu trúc cây)

1) *Xuất ra theo thứ tự : giữa, đầu, cuối* các phần tử trên cây nhị phân sau:



2) a. *Tìm cây nhị phân thỏa đồng thời hai điều kiện kết xuất sau:*

theo thứ tự đầu NLR của nó là dãy ký tự sau:

A, B, C, D, E, Z, U, T, Y

và theo thứ tự giữa LNR của nó là dãy ký tự sau:

D, C, E, B, A, U, Z, T, Y

b. (*) Khi cho trước 2 trong 3 kết quả duyệt NLR, LNR, LRN thì có luôn xác định duy nhất cây nhị phân thỏa điều kiện nêu ra không ? Dùng chương trình để kiểm chứng ?

3) a. *Biểu diễn mỗi biểu thức số học dưới đây trên cây nhị phân, từ đó rút ra dạng biểu thức hậu tố* của chúng:

i. a/(b*c)

ii. $a^5 + 4a^3 - 3a^2 + 7$

iii. (a+b)*(c-d)

iv. S^{a+b}

b. (*) Viết thuật toán và chương trình:

- Chuyển một biểu thức số học ký hiệu lên cây nhị phân (có kiểm tra biểu thức đã cho có hợp cú pháp không ?).
- Xuất ra biểu thức số học đó dưới dạng: trung tố, hậu tố, tiền tố.
- Sau đó nhập trị cho các ký hiệu trong biểu thức, hãy đánh giá biểu thức hậu tố tương ứng.

4) *Xây dựng cây tìm kiếm nhị phân BST và cây AVL từ mỗi bộ mục dữ liệu đầu vào* như sau:

a. 1, 2, 3, 4, 5

b. 5, 4, 3, 2, 1

c. fe, cx, jk, ha, gc, ap, aa, by, my, da

d. 8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 10, 12, 17, 16, 18.
Sau đó xóa lần lượt các nút sau: 2, 10, 19, 8, 20, 6, 1.

5) *Viết một chương trình có các tác dụng sau:*

a. Nhập từ bàn phím các số nguyên vào một cây nhị phân tìmkiếm (BST) mà nút gốc được trỏ bởi con trỏ Root.

b. Xuất các phần tử trên cây BST trên theo thứ tự : đầu, giữa, cuối theo dòng và vẽ sơ đồ cây (*) (chỉ yêu cầu trường hợp khi số phần tử của cây nhị phân không quá lớn !).
c. Tìm và xóa (nếu có thể) phần tử trên cây Root có dữ liệu trùng với một mục dữ liệu Item cho trước được nhập từ bàn phím.
d. Sắp xếp n mục dữ liệu (được cài đặt bằng mảng hay DSLK) bằng phương pháp cây nhị phân tìm kiếm BSTSort.

Yêu cầu: viết các thao tác trên bằng 2 phương pháp: đệ quy và lặp (*).
(**) Riêng với duyệt cây, hãy viết dưới dạng lặp cả 3 phương pháp duyệt trong một hàm duy nhất có tính khái quát.

e. Kiểm tra lại kết quả của bài tập 4) bằng chương trình vừa xây dựng.

6) Tương tự bài tập 5, nhưng mỗi nút có thêm trường con trỏ Parent chỉ đến nút cha của nó.

7) (*) Xây dựng các thao tác cơ bản trên cây n-phân được biểu diễn bởi cây nhị phân: chèn một nút, tạo cây n-phân, xóa một nút, hủy cây n-phân.

8) Cho cây nhị phân T. Viết chương trình chứa các hàm có tác dụng xác định:
a. *Tổng số nút* của cây. Số nút tối đa của cây nhị phân có chiều cao h là bao nhiêu? Chứng minh điều khẳng định bằng qui nạp và kiểm nghiệm lại bằng chương trình.
b. (*) Số nút của cây ở mức k. Số nút tối đa ở mức k của cây nhị phân là bao nhiêu ? Chứng minh điều khẳng định bằng qui nạp và kiểm nghiệm lại bằng chương trình.
c. Số nút lá.
d. (*) *Chiều cao* của cây.
e. *Tổng giá trị trường dữ liệu (số !)* trên các nút của cây.
f. *Kiểm tra xem nó có phải là một cây nhị phân chặt* (là cây nhị phân mà mỗi nút khác nút lá đều có đúng 2 con) hay không ?
g. *Kiểm tra xem T có phải là cây cân bằng hoàn toàn* hay không ?
h. Số nút có đúng 2 con khác rỗng
i. Số nút có đúng 1 con khác rỗng
j. Số nút có khóa nhỏ hơn x trên cây nhị phân hoặc cây BST
k. Số nút có khóa lớn hơn x trên cây nhị phân hoặc cây BST
l. Số nút có khóa nhỏ hơn x và lớn hơn y ($y \le x$) trên cây nhị phân hoặc cây BST
m. Duyệt cây theo chiều rộng
n. Duyệt cây theo chiều sâu
o. Độ lệch lớn nhất của các nút trên cây (độ lệch của một nút là trị tuyệt đối của hiệu số giữa chiều cao của cây con phải và cây con trái của nó)
p. Đảo nhánh trái và phải của mọi nút trên một cây nhị phân

Yêu cầu: viết các thao trên bằng 2 phương pháp: đệ quy và lặp (*).

9) *Viết chương trình xây dựng cây nhị phân tìm kiếm có chiều cao bé nhất* từ một dãy có thứ tự tăng của các phần tử được lưu trữ trên một danh sách liên kết.

10) *a. Hãy vẽ cây AVL có chiều cao cực đại có 12 nút*

b. Hãy tìm một ví dụ về một cây AVL có chiều cao là 6 và khi hủy một nút lá (chỉ ra cụ thể), việc cân bằng lại cây lan truyền lên tận gốc.

c. (*) Viết chương trình thể hiện các thao tác cơ bản trên cây AVL: chèn một nút, xóa một nút, tạo cây AVL, hủy cây AVL. Kiểm tra lại bằng chương trình với dữ liệu của câu a. và b. trên đây.

11) (*) Viết chương trình cho phép tạo, thêm, bớt, tra cứu, sửa chữa từ điển.

TÀI LIỆU THAM KHẢO

[1] A.V. AHO , J.E. HOPCROFT , J.D. ULMANN: Data structures and algorithms. Addition Wesley - 1983.

[2] DONALD KNUTH: The Art of Programming. (vol.1: Fundamental Algorithms, vol. 3: Sorting and Searching). Addition Wesley Puplishing Company - 1973.

[3] ĐINH MẠNH TƯỜNG: Cấu trúc dữ liệu và giải thuật. NXB KHKT - 2001.

[4] ĐỖ XUÂN LÔI: Cấu trúc dữ liệu và giải thuật. NXB KHKT - 1995.

[5] LARRY N. HOFF, SANFORD LEESTMA: Lập trình nâng cao bằng Pascal với các cấu trúc dữ liệu. Bản dịch của Lê Minh Trung. Công ty Scitec - 1991.

[6] NGUYỄN TRUNG TRỰC: Cấu trúc dữ liệu. Trung tâm điện toán, trường ĐH Bách khoa TP. HCM – 1992.

[7] NIKLAUS WIRTH: Cấu trúc dữ liệu + Giải thuật = Chươngtrình (Nguyễn Quốc Cường dịch). NXB ĐH và THCN – 1991

[8] TRẦN HẠNH NHI & DƯƠNG ANH ĐỨC: Nhập môn cấu trúc dữ liệu và giải thuật. Khoa Công nghệ thông tin, ĐH KHTN TP HCM – 2000.