BỘ GIÁO DỤC VÀ ĐÀO TẠO

ĐẠI HỌC CÔNG NGHỆ TP.HCM

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Biên Soạn:

TS. Nguyễn Thị Hải Bình

www.hutech.edu.vn

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Các ý kiến đóng góp về tài liệu học tập này, xin gửi về e-mail của ban biên tập:

tailieuhoctap@hutech.edu.vn

MỤC LỤC

DANH MỤC HÌNH ẢNH

HƯỚNG DẪN

MÔ TẢ MÔN HỌC

Cấu trúc dữ liệu và giải thuật là môn học trình bày các phương pháp tổ chức và những thao tác cơ sở trên từng cấu trúc dữ liệu, kết hợp với việc phát triển tư duy giải thuật để hình thành nên chương trình máy tính.

Môn học này giúp học viên hiểu được tầm quan trọng của giải thuật và cách tổ chức dữ liệu, là hai thành tố quan trọng nhất cho một chương trình. Ngoài ra, môn học này còn giúp học viên củng cố và phát triển kỹ năng lập trình được học trong giai đoạn trước.

NỘI DUNG MÔN HỌC

- Bài 1. TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT. Bài này cung cấp cho học viên khái niệm về cấu trúc dữ liệu và giải thuật, phân tích mối quan hệ giữa cấu trúc dữ liệu và giải thuật có ảnh hưởng đến đề án tin học như thế nào. Ngoài ra, còn trình bày cách đánh giá độ phức tạp của giải thuật.
- Bài 2. CẤU TRÚC DANH SÁCH. Bài này trình bày cấu trúc dữ liệu danh sách, các phương pháp cài đặt danh sách. Trong đó, chú trọng vào kiểu dữ liệu danh sách liên kết.
- Bài 3. CẤU TRÚC STACK VÀ QUEUE. Bài này trình bày cấu trúc dữ liệu Stack và Queue, các thao tác cơ bản trên Stack và Queue, và cách cài đặt. Ngoài ra, bài còn đưa ra một số bài toán ứng dụng thực tế của Stack và Queue.
- Bài 4. CẤU TRÚC CÂY, CÂY NHỊ PHÂN, CÂY NHỊ PHÂN TÌM KIẾM. Bài này trình bày cấu trúc dữ liệu cây và cung cấp cho học viên các khái niệm liên quan đến cây, các loại cây. Trong đó, mô tả cụ thể cây nhị phân và cách hiện thực nó. Bài này cũng trình bày cấu trúc cây nhị phân tìm kiếm với các khái niệm, thao tác trên cây nhị phân tìm kiếm và cách cài đặt cây nhị phân tìm kiếm.
- Bài 5. CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG - AVL. Bài này cung cấp khái niệm cây cân bằng, từ đó đưa ra cấu trúc cây AVL và các tác vụ cơ bản trên nó.

- Bài 6. ĐỒ THỊ VÀ BIỂU DIỄN ĐỒ THỊ. Bài này trình bày về khái niệm đồ thị, đồ thị vô hướng, đồ thị có hướng, đồ thị có trọng số, và một số dạng đồ thị đặc biệt khác. Bài học cũng cung cấp các khái niệm cơ bản trong đồ thị như đỉnh kề, cạnh (cung) kề, bậc của đỉnh, tính liên thông của đồ thị. Ngoài ra, bài học mô tả các cách biểu diễn đồ thị, bao gồm: ma trận kề, danh sách cạnh (cung), và danh sách kề.
- Bài 7. ĐƯỜNG ĐI VÀ CHU TRÌNH. Bài này cung cấp khái niệm về đường đi và chu trình trong đồ thị. Bài học trình bày về hai dạng đường và chu trình đặc biệt, đó là đường Euler, chu trình Euler, đường Hamilton, và chu trình Hamilton. Bài học cũng cung cấp các giải thuật để tìm đường (chu trình) Euler và Hamilton, cũng như một số ứng dụng của hai loại chu trình đặc biệt này.
- Bài 8. DUYỆT ĐỒ THỊ. Bài học trình bày hai giải thuật cơ bản để duyệt tất cả các đỉnh của đồ thị, đó là giải thuật duyệt đồ thị theo chiều rộng và duyệt đồ thị theo chiều sâu. Hai bài toán ứng dụng của giải thuật duyệt đồ thị là bài toán tìm đường đi giữa hai đỉnh của đồ thị và bài toán xác định các thành phần liên thông của đồ thị cũng được trình bày trong bài học này.
- Bài 9. CÂY BAO TRÙM VÀ CÂY BAO TRÙM NHỎ NHẤT. Bài này trình bày khái niệm về cây, cây bao trùm, và cây bao trùm nhỏ nhất của đồ thị. Bài học cũng trình bày hai giải thuật để tìm cây bao trùm nhỏ nhất của đồ thị, bao gồm giải thuật Kruskal và giải thuật Prim.
- Bài 10. ĐƯỜNG ĐI NGẮN NHẤT. Bài này cung cấp kiến thức về bài toán tìm đường đi ngắn nhất trong đồ thị. Ba giải thuật tìm đường đi ngắn nhất được trình bày trong bài học bao gồm giải thuật Dijstra, giải thuật Bellman-Ford, và giải thuật Floyd.

KIẾN THỨC TIỀN ĐỀ

Môn học Cấu trúc dữ liệu và Giải thuật đòi hỏi sinh viên có nền tảng về Tin học đại cương và Lập trình căn bản.

YÊU CẦU MÔN HỌC

Người học phải dự học đầy đủ các buổi lên lớp và làm bài tập đầy đủ ở nhà.

CÁCH TIẾP NHẬN NỘI DUNG MÔN HỌC

Để học tốt môn này, người học cần tập trung tiếp thu bài giảng tại lớp, làm tất cả các ví dụ và bài tập; giờ thực hành cần nắm và cài đặt được các cấu trúc dữ liệu và thuật toán đã học.

Đối với mỗi bài học, nên đọc trước mục tiêu và tóm tắt bài học, sau đó đọc nội dung bài học.

PHƯƠNG PHÁP ĐÁNH GIÁ MÔN HỌC

Môn học được đánh giá gồm:

- Điểm quá trình: 50%. Hình thức và nội dung do GV quyết định, phù hợp với quy chế đào tạo và tình hình thực tế tại nơi tổ chức học tập.
- Điểm thi: 50%. Hình thức bài thi tự luận. Nội dung gồm các bài tập liên quan đến kiến thức thuộc bài thứ 1 đến bài thứ 10.

BÀI 1: TỔNG QUAN VỀ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Sau khi học xong bài này, học viên có thể:

- *Nắm được vai trò của việc tổ chức dữ liệu trong một đề án tin học*
- *Thấy rõ mối quan hệ giữa giải thuật và cấu trúc dữ liệu*
- *Biết các yêu cầu tổ chức cấu trúc dữ liệu*
- *Cách đánh giá độ phức tạp của giải thuật*

1.1 VAI TRÒ CỦA CẤU TRÚC DỮ LIỆU TRONG MỘT ĐỀ ÁN TIN HỌC

Thực hiện một đề án tin học là chuyển bài toán thực tế thành bài toán có thể giải quyết trên máy tính. Một bài toán thực tế bất kỳ đều bao gồm các đối tượng dữ liệu và các yêu cầu xử lý trên những đối tượng đó. Vì vậy, để xây dựng một mô hình tin học phản ánh được bài toán thực tế cần chú trọng hai vấn đề:

- *Tổ chức biểu diễn các đối tượng thực tế:* Các thành phần dữ liệu thực tế đa dạng, phong phú và thường chứa đựng những quan hệ nào đó với nhau. Do đó, trong mô hình tin học của bài toán, cần phải tổ chức xây dựng các cấu trúc thích hợp nhất sao cho vừa có thể phản ánh chính xác các dữ liệu thực tế này, vừa có thể dễ dàng dùng máy tính để xử lý. Công việc này được gọi là *xây dựng cấu trúc dữ liệu cho bài toán*.
- *Xây dựng các thao tác xử lý dữ liệu:* Từ những yêu cầu xử lý thực tế, cần tìm ra các giải thuật tương ứng để xác định trình tự các thao tác máy tính phải thi hành để cho ra kết quả mong muốn, đây là bước *xây dựng giải thuật cho bài toán*.

Tuy nhiên, khi giải quyết một bài toán trên máy tính, chúng ta thường có khuynh hướng chỉ chú trọng đến việc xây dựng giải thuật mà quên đi tầm quan trọng của việc

tổ chức dữ liệu trong bài toán. Giải thuật phản ánh các phép xử lý, còn đối tượng xử lý của giải thuật lại là dữ liệu, chính dữ liệu chứa đựng các thông tin cần thiết để thực hiện giải thuật. Để xác định được giải thuật phù hợp cần phải biết nó tác động đến loại dữ liệu nào và khi chọn lựa cấu trúc dữ liệu cũng cần phải hiểu rõ những thao tác nào sẽ tác động đến nó. Như vậy trong một đề án tin học, giải thuật và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau, được thể hiện qua công thức:

Cấu trúc dữ liệu + Giải thuật = Chương trình

Với một cấu trúc dữ liệu đã chọn, sẽ có những giải thuật tương ứng, phù hợp. Khi cấu trúc dữ liệu thay đổi thường giải thuật cũng phải thay đổi theo để tránh việc xử lý gượng ép, thiếu tự nhiên trên một cấu trúc không phù hợp. Hơn nữa, một cấu trúc dữ liệu tốt sẽ giúp giải thuật xử lý trên đó có thể phát huy tác dụng tốt hơn, vừa đáp ứng nhanh vừa tiết kiệm vật tư, giải thuật cũng dễ hiểu và đơn giản hơn.

1.2 CÁC TIÊU CHUẨN ĐÁNH GIÁ CẤU TRÚC DỮ LIỆU

Một cấu trúc dữ liệu tốt phải thỏa mãn các tiêu chuẩn sau:

Phản ánh đúng thực tế: Đây là tiêu chuẩn quan trọng nhất, quyết định tính đúng đắn của toàn bộ bài toán. Cần xem xét kỹ lưỡng cũng như dự trù các trạng thái biến đổi của dữ liệu trong chu trình sống để có thể chọn cấu trúc dữ liệu lưu trữ thể hiện chính xác đối tượng thực tế.

Phù hợp với các thao tác trên đó: Tiêu chuẩn này giúp tăng tính hiệu quả của đề án: việc phát triển các thuật toán đơn giản, tự nhiên hơn; chương trình đạt hiệu quả cao hơn về tốc độ xử lý.

Tiết kiệm tài nguyên hệ thống: Cấu trúc dữ liệu chỉ nên sử dụng tài nguyên hệ thống vừa đủ để đảm nhiệm được chức năng của nó.Thông thường có 2 loại tài nguyên cần lưu tâm nhất: CPU và bộ nhớ. Tiêu chuẩn này nên cân nhắc tùy vào tình huống cụ thể khi thực hiện đề án. Nếu tổ chức sử dụng đề án cần có những xử lý nhanh thì khi chọn cấu trúc dữ liệu yếu tố tiết kiệm thời gian xử lý phải đặt nặng hơn tiêu chuẩn sử dụng tối ưu bộ nhớ, và ngược lại.

1.3 TRỪU TƯỢNG HOÁ DỮ LIỆU

Trừu tượng hoá là ý niệm về sự vật hay hiện tượng sau khi thu thập chắt lọc những thông tin có ý nghĩa; và loại bỏ đi những thông tin không cần thiết hoặc những chi tiết không quan trọng. Thông tin bao gồm các trạng thái tĩnh (data) và các tác vụ (operation) lên dữ liệu đó.

Sự trừu tượng hoá bao hàm trừu tượng hoá dữ liệu để thu thập thông tin về dữ liệu và trừu tượng hoá tác vụ để thu thập các tác vụ liên quan. Kết quả của quá trình trừu tượng hoá giúp chúng ta xây dựng một mô hình cho một kiểu dữ liệu mới gọi là kiểu dữ liệu trừu tượng (Abstract Data Type - ADT), mỗi kiểu dữ liệu trừu tượng có mô tả dữ liệu và các tác vụ liên quan.

Ví dụ 1.1: mô tả kiểu dữ liệu trừu tượng về số hữu tỉ a/b với các tác vụ cộng hai số hữu tỉ, nhân hai số hữu tỉ, chia hai số hữu tỉ.

KIỂU DỮ LIỆU TRỪU TƯỢNG VỀ SỐ HỮU TỈ

Mô tả dữ liệu:

- Tử số.
- Mẫu số (mẫu số phải khác 0).

Mô tả tác vụ:

- Tác vụ cộng: add(sốhữutỉ1, sốhữutỉ2)

  Đầu vào:

  a,b là tử và mẫu của sốhữutỉ1

  c,d là tử và mẫu của sốhữutỉ2

  Đầu ra:

  ad+bc là tử của số hữu tỉ kết quả

  bd là mẫu của số hữu tỉ kết quả.
- Tác vụ nhân: mul(sốhữutỉ1, sốhữutỉ2)

  Đầu vào: ...

  Đầu ra: ...

...

1.4 KIỂU DỮ LIỆU CƠ BẢN

Các loại dữ liệu cơ bản thường là các loại dữ liệu đơn giản, không có cấu trúc. Chúng thường là các giá trị vô hướng như các số nguyên, số thực, các ký tự, các giá

trị logic... Các loại dữ liệu này, do tính thông dụng và đơn giản của mình, thường được các ngôn ngữ lập trình (NNLT) cấp cao xây dựng sẵn như một thành phần của ngôn ngữ để giảm nhẹ công việc cho người lập trình. Chính vì vậy đôi khi người ta còn gọi chúng là các kiểu dữ liệu định sẵn.

Thông thường, các kiểu dữ liệu cơ bản bao gồm:

Kiểu có thứ tự rời rạc: số nguyên, ký tự, logic, liệt kê, miền con...

Kiểu không rời rạc: số thực

Các kiểu dữ liệu định sẵn trong C gồm các kiểu sau:

Tên kiểu	Kích thước	Miền giá trị	Ghi chú
char	1 byte	-128 đến 127	Có thể dùng như số nguyên 1-byte có dấu hoặc kiểu ký tự
unsigned char	1 byte	0 đến 255	Số nguyên 1-byte không dấu
int	2 bytes	-32,738 đến 32,767	short int
unsigned int	2 bytes	0 đến 65335	unsigned short int
int	4 bytes	-2^{32} đến 2^{31} -1	long int (x64)
long	4 bytes	-2^{32} đến 2^{31} -1	
unsigned long	4 bytes	0 đến 2^{32}-1	
float	4 bytes	3.4E-38 đến 3.4E38	Giới hạn chỉ trị tuyệt đối. Các giá trị bé hơn 3.4E-38 được coi bằng 0. Tuy nhiên kiểu float chỉ có 7 chữ số có nghĩa.
double	8 bytes	1.7E-308 đến 1.7E308	

1.5 KIỂU DỮ LIỆU CÓ CẤU TRÚC

Trong nhiều trường hợp, các kiểu dữ liệu cơ sở không đủ để phản ánh tự nhiên và đầy đủ bản chất của sự vật thực tế, dẫn đến nhu cầu phải xây dựng các kiểu dữ liệu mới dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu đã được định nghĩa. Những kiểu dữ liệu được xây dựng như thế gọi là kiểu dữ liệu có cấu trúc. Đa số các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như

mảng, chuỗi, tập tin, bản ghi... và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

Ví dụ 1.2: Để mô tả một đối tượng sinh viên, cần quan tâm đến các thông tin sau:

- Mã sinh viên: chuỗi ký tự
- Tên sinh viên: chuỗi ký tự
- Ngày sinh: kiểu ngày tháng
- Nơi sinh: chuỗi ký tự
- Điểm thi: số nguyên

Các kiểu dữ liệu cơ sở cho phép mô tả một số thông tin như:

```
int Diemthi;
```

Các thông tin khác đòi hỏi phải sử dụng các kiểu có cấu trúc như:

```
char masv[15];
char tensv[15];
char noisinh[15];
```

Để thể hiện thông tin về ngày tháng năm sinh cần phải xây dựng một kiểu bản ghi,

```
typedef struct tagDate{
        char ngay;
        char thang;
        char thang;
}Date;
```

Cuối cùng, ta có thể xây dựng kiểu dữ liệu thể hiện thông tin về một sinh viên:

```
typedef struct tagSinhVien{
        char masv[15];
        char tensv[15];
        Date ngaysinh;
        char noisinh[15];
        int Diem thi;
}SinhVien;
```

Giả sử đã có cấu trúc phù hợp để lưu trữ một sinh viên, nhưng thực tế lại cần quản lý nhiều sinh viên, lúc đó nảy sinh nhu cầu xây dựng kiểu dữ liệu mới...Mục tiêu của việc nghiên cứu cấu trúc dữ liệu chính là tìm những phương cách thích hợp để tổ

chức, liên kết dữ liệu, hình thành các kiểu dữ liệu có cấu trúc từ những kiểu dữ liệu đã được định nghĩa.

1.6 ĐỘ PHỨC TẠP CỦA GIẢI THUẬT

1.6.1 Khái niệm về độ phức tạp của giải thuật

Thời gian máy tính thực hiện một giải thuật không chỉ phụ thuộc vào bản thân giải thuật đó, mà còn tùy thuộc từng máy tính. Để đánh giá hiệu quả của một giải thuật, có thể xét số các phép tính phải thực hiện khi thực hiện giải thuật đó. Thông thường số các phép tính được thực hiện phụ thuộc vào cỡ của bài toán, tức là độ lớn của đầu vào. Vì thế độ phức tạp giải thuật là một hàm phụ thuộc đầu vào. Tuy nhiên trong những ứng dụng thực tiễn, chúng ta không cần biết chính xác hàm này mà chỉ cần biết một ước lượng đủ tốt của chúng.

Thời gian thực hiện chương trình:

Thời gian thực hiện một chương trình là một hàm của kích thước dữ liệu vào, ký hiệu T(n) trong đó n là kích thước (độ lớn) của dữ liệu vào. Chương trình tính tổng của n số có thời gian thực hiện là T(n) = C*n trong đó C là một hằng số. Thời gian thực hiện chương trình là một hàm không âm, tức là $T(n) \geq 0$ với mọi $n \geq 0$.

Đơn vị đo thời gian thực hiện:

Đơn vị của T(n) không phải là đơn vị đo thời gian bình thường như giờ, phút giây ... mà thường được xác định bởi số các lệnh được thực hiện trong một máy tính lý tưởng. Khi ta nói thời gian thực hiện của một chương trình là $T(n) = C * n$ thì có nghĩa là chương trình ấy cần $C * n$ chỉ thị thực thi.

Thời gian thực hiện trong trường hợp xấu nhất:

Nói chung thì thời gian thực hiện chương trình không chỉ phụ thuộc vào kích thước mà còn phụ thuộc vào tính chất của dữ liệu vào. Nghĩa là dữ liệu vào có cùng kích thước nhưng thời gian thực hiện chương trình có thể khác nhau. Chẳng hạn chương trình sắp xếp dãy số nguyên tăng dần, khi ta cho vào dãy có thứ tự thì thời gian thực hiện khác với khi ta cho vào dãy chưa có thứ tự, hoặc khi ta cho vào một dãy đã có thứ tự tăng thì thời gian thực hiện cũng khác so với khi ta cho vào một dãy đã có thứ

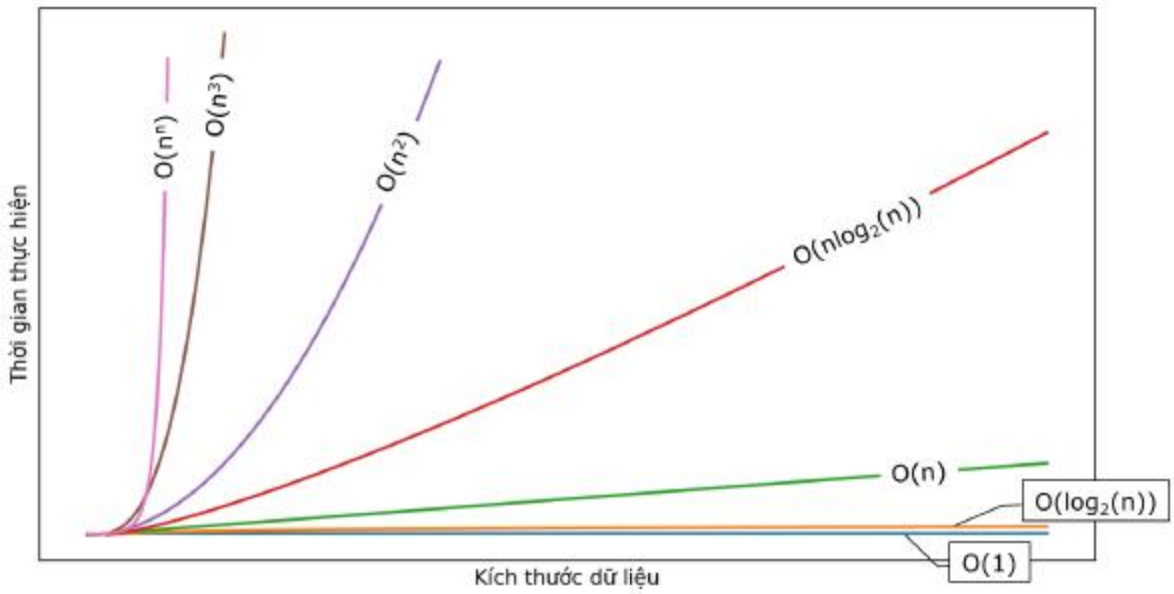
tự giảm. Vì vậy thường ta coi T(n) là thời gian thực hiện chương trình trong trường hợp xấu nhất trên dữ liệu vào có kích thước n, tức là: T(n) là thời gian lớn nhất để thực hiện chương trình đối với mọi dữ liệu vào có cùng kích thước n.

Độ phức tạp của giải thuật:

Cho một hàm T(n), T(n) gọi là độ phức tạp f(n) nếu tồn tại các hằng số C và N_0 sao cho $T(n) \leq C*f(n)$ với mọi $n \geq N_0$ và ký hiệu là O(f(n)).

Nói cách khác độ phức tạp tính toán của giải thuật là một hàm chặn trên của hàm thời gian. Vì hằng nhân tử C trong hàm chặn trên không có ý nghĩa nên ta có thể bỏ qua vì vậy hàm thể hiện độ phức tạp có các dạng thường gặp sau:

Ký hiệu	Ý nghĩa
O(1)	Nếu T(n) là hằng số (T(n)=C)
$O(\log_2 n)$	Độ phức tạp dạng logarit
O(n)	Độ phức tạp tuyến tính
$O(n\log_2 n)$	Độ phức tạp tuyến tính logarit
$O(n^2)$, $O(n^3)$, ..., $O(n^\alpha)$	Độ phức tạp đa thức
O(n!), $O(n^n)$	Độ phức tạp dạng hàm mũ



Hình 1.1: Biểu đồ thể hiện độ phức tạp của giải thuật

Một giải thuật mà thời gian thực hiện có độ phức tạp là một hàm đa thức thì chấp nhận được tức là có thể cài đặt để thực hiện, còn các giải thuật có độ phức tạp hàm mũ thì phải tìm cách cải tiến giải thuật.

1.6.2 Cách tính độ phức tạp của giải thuật

Quy tắc bỏ hằng số: Nếu đoạn chương trình có thời gian thực hiện là O(c.f(n)) với c là một hằng số dương, thì có thể coi O(c.f(n)) = O(f(n)).

Quy tắc lấy max: Nếu đoạn chương trình có thời gian thực hiện là T(n) = O(f(n) + g(n)) thì có thể coi T(n) = max(f(n), g(n)).

Quy tắc cộng: Nếu đoạn chương trình P1 có thời gian thực hiện là f(n) và đoạn chương trình P2 có thời gian thực hiện là g(n), thì thời gian để thực hiện tuần tự hai đoạn chương trình P1 và P2 là T(n) = O(f(n) + g(n)).

Quy tắc nhân: Nếu đoạn chương trình P1 có thời gian thực hiện là f(n) và đoạn chương trình P2 có thời gian thực hiện là g(n), thì thời gian để thực hiện hai đoạn chương trình đó lồng nhau là T(n) = O(f(n)×g(n)).

Một số công thức thường dùng:

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=a}^{b} 1 = \sum_{i=1}^{b} 1 - \sum_{i=1}^{a} 1 = b - a + 1$$

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

$$\sum_{i=1}^{n^2} i = \frac{n^2(n^2+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

TÓM TẮT

Trong bài này, học viên cần nắm:

Vai trò của việc tổ chức dữ liệu trong một đề án tin học: Tổ chức dữ liệu chính là tổ chức biểu diễn các đối tượng thực tế của bài toán. Vì vậy, tổ chức dữ liệu đóng vai trò quan trọng trong việc giải quyết xử lý bài toán, ảnh hưởng đến kết quả đạt được của đề án tin học. Tổ chức dữ liệu là công việc xây dựng cấu trúc dữ liệu cho bài toán. Một cấu trúc dữ liệu tốt phải thỏa mãn các tiêu chuẩn sau: Phản ánh đúng thực tế, phù hợp với các thao tác trên đó, tiết kiệm tài nguyên hệ thống.

Mối quan hệ giữa giải thuật và cấu trúc dữ liệu:

Cấu trúc dữ liệu + Giải thuật = Chương trình.

Phân biệt được kiểu dữ liệu cơ bản và kiểu dữ liệu có cấu trúc: Kiểu dữ liệu cơ bản trong C gồm: char, unsigned char, int, unsigned int, long, unsigned long, float, double, long double. Kiểu dữ liệu có cấu trúc là kiểu dữ liệu được xây dựng dựa trên việc tổ chức, liên kết các thành phần dữ liệu có kiểu dữ liệu đã được định nghĩa. Các ngôn ngữ lập trình đều cài đặt sẵn một số kiểu có cấu trúc cơ bản như mảng, chuỗi, tập tin, bản ghi... và cung cấp cơ chế cho lập trình viên tự định nghĩa kiểu dữ liệu mới.

Khái niệm và cách xác định độ phức tạp của giải thuật.

CÂU HỎI ÔN TẬP

Câu 1: Viết chương trình C khai báo kiểu dữ liệu là mảng một chiều, chương trình có các chức năng như sau:

a. Nhập giá trị vào mảng.

b. Sắp xếp mảng theo thứ tự từ nhỏ đến lớn.

c. Xem nội dung các phần tử trong mảng.

Câu 2: Viết chương trình C có khai báo kiểu dữ liệu là mảng hai chiều, chương trình có các chức năng sau:

a. Nhập giá trị vào ma trận.

b. Nhân hai ma trận thành ma trận tích

c. Xem nội dung của các phần tử trong ma trận.

Câu 3: Hãy xây dựng và hiện thực kiểu dữ liệu trừu tượng cho thông tin sinh viên với các thao tác nhập, xuất thông tin của sinh viên.

Câu 4: Hãy xây dựng và hiện thực kiểu dữ liệu trừu tượng của số hữu tỉ a/b với các tác vụ cộng hai số hữu tỉ, nhân hai số hữu tỉ, chia hai số hữu tỉ.

Câu 5: Hãy xây dựng và hiện thực kiểu dữ liệu trừu tượng cho số phức với các tác vụ cộng, trừ, nhân, chia hai số phức.

BÀI 2: DANH SÁCH

Sau khi học xong bài này, học viên có thể:

- *Hiểu được cấu trúc danh sách.*
- *Cài đặt danh sách theo hai phương pháp: cài đặt danh sách theo kiểu kế tiếp và kiểu liên kết.*
- *Hiện thực danh sách kề và danh sách liên kết, áp dụng vào bài toán thực tế.*

2.1 KHÁI NIỆM

Danh sách (List) là một trong những cấu trúc cơ bản nhất được cài đặt trong hầu hết các chương trình ứng dụng. Danh sách là một kiểu dữ liệu trừu tượng gồm nhiều nút cùng kiểu dữ liệu, các nút trong danh sách có thứ tự.

Có hai cách cài đặt danh sách:

- Cài đặt theo kiểu kế tiếp, ta có danh sách kề
- Cài đặt theo kiểu liên kết, ta có danh sách liên kết.

2.2 CẤU TRÚC DANH SÁCH

Mô tả dữ liệu:

Danh sách là một tập hợp các nút cùng kiểu dữ liệu, các nút trong danh sách có thứ tự.

Mô tả các tác vụ:

- Tác vụ *init*

Chức năng: khởi tạo danh sách ban đầu rỗng, chưa có nút.

Dữ liệu nhập: không.

Tác vụ *isEmpty*:

Chức năng: kiểm tra danh sách có rỗng không.

Dữ liệu nhập: không.

Dữ liệu xuất: TRUE|FALSE

- Tác vụ *isFull*:

Chức năng: kiểm tra danh sách có bị đầy không.

Dữ liệu nhập: không.

Dữ liệu xuất: TRUE|FALSE.

- Tác vụ *ListSize*:

Chức năng: kiểm tra số nút có trong danh sách.

Dữ liệu nhập: không.

Dữ liệu xuất: số nút trong danh sách.

- Tác vụ *Retrieve*:

Chức năng: truy xuất nút tại vị trí *position* trong danh sách.

Dữ liệu nhập: *pos* là vị trí của nút cần truy xuất trong danh sách.

Điều kiện: *0=<pos<=num - 1* (*num* là số nút của danh sách)

- Tác vụ *Insert*:

Chức năng: thêm nút vào vị trí *pos* của danh sách.

Dữ liệu nhập: nút mới và vị trí *pos* (vị trí thêm nút mới).

Điều kiện: *0=<pos<=num*.

Dữ liệu xuất: không.

- Tác vụ *Remove*:

Chức năng: Xóa nút tại vị trí *pos* của danh sách.

Dữ liệu nhập: *pos* (vị trí của nút xóa).

Điều kiện: *0=<pos<=num – 1*

Dữ liệu xuất: nút bị xóa.

- Tác vụ *Replace*:

  Chức năng: thay thế nút tại vị trí *pos* của danh sách bằng nút khác.

  Dữ liệu nhập: nút khác và vị trí thay thế *pos*.

  Điều kiện: *0=<pos<=num-1*

  Dữ liệu xuất: không

- Tác vụ *ShowList*:

  Chức năng: duyệt tất cả các nút của danh sách.

  Dữ liệu nhập: không.

  Dữ liệu xuất: không.

- Tác vụ *Sort*:

  Chức năng: sắp xếp lại danh sách theo một khoá sắp xếp.

  Dữ liệu nhập: key (khóa sắp xếp)

  Dữ liệu xuất: không.

- Tác vụ *Search*:

  Chức năng: tìm kiếm một nút trong danh sách theo một khoá tìm kiếm.

  Dữ liệu nhập: key là khóa cần tìm.

  Dữ liệu xuất: TRUE|FALSE và pos. TRUE là tìm thấy key trong danh sách và pos chỉ vị trí tìm thấy.

- Tác vụ *Clearlist*:

  Chức năng: xoá danh sách.

  Dữ liệu nhập: không

  Dữ liệu xuất: không.

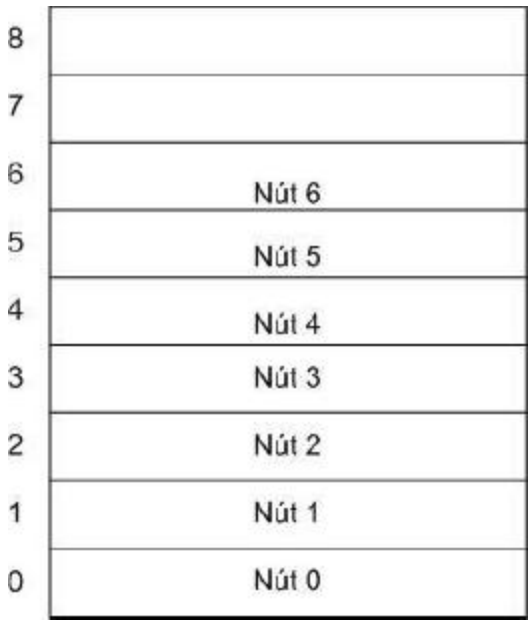
2.3 PHƯƠNG PHÁP CÀI ĐẶT DANH SÁCH

Có hai cách cài đặt danh sách: cài đặt theo kiểu danh sách kế tiếp và cài đặt theo kiểu danh sách liên kết.

Cài đặt theo kiểu kế tiếp:

Cài đặt theo kiểu kế tiếp sẽ bố trí các nút trong danh sách liên kết kế cận nhau trong bộ nhớ, cài đặt kiểu này tạo nên danh sách kề. Mảng, chuỗi ký tự, stack hay hàng đợi cài đặt theo kiểu kế tiếp, ... là những dạng khác nhau của danh sách kề.

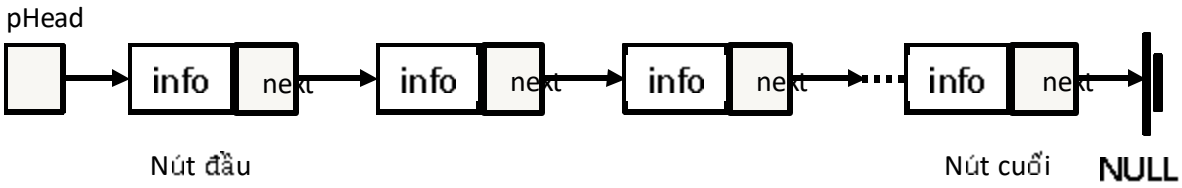
Hình sau đây minh họa danh sách kề dùng mảng 1 chiều, mỗi phần tử trên mảng là một nút của danh sách, danh sách hiện có 7 nút trải dài từ nút 0 đến nút 6 của mảng.



Hình 2.1: Danh sách kề dùng mảng một chiều.

Cài đặt theo kiểu liên kết:

Danh sách được cài đặt theo kiểu liên kết gọi là danh sách liên kết. Mỗi nút trong danh sách có trường ***info*** là nội dung của nút và trường ***next*** là con trỏ chỉ nút kế tiếp trong danh sách. Con trỏ đầu của danh sách (*pHead*) chỉ nút đầu tiên, nút cuối cùng của danh sách có trường next trỏ đến vị trí **NULL**. Hình vẽ sau minh hoạ cách cài đặt bằng danh sách liên kết:



Hình 2.2: Danh sách liên kết.

So sánh hai kiểu cài đặt:

Danh sách kề nếu khai báo kích thước danh sách phù hợp thì danh sách kề tối ưu về bộ nhớ vì tại mỗi nút sẽ không cần chứa trường next. Và tốc độ truy xuất phần tử thứ *i* trong danh sách kề rất nhanh do các phần tử được truy cập ngẫu nhiên.

Tuy nhiên, về số nút cấp phát cho danh sách kề là cố định nên số nút cần dùng lúc thừa, lúc thiếu. Hơn nữa, danh sách kề bị hạn chế khi thực hiện các tác vụ *insert, remove* vì mỗi khi thực hiện các tác vụ này chúng ta phải dời chỗ rất nhiều nút. Số nút của danh sách càng lớn thì số lần dời chỗ càng nhiều nên càng chậm.

Số nút cấp phát cho danh sách liên kết thay đổi khi chương trình đang chạy nên việc cấp phát nút cho danh sách liên kết rất linh hoạt: khi nào cần thì cấp phát nút, khi không cần thì giải phóng nút. Danh sách liên kết rất thích hợp khi hiện thực các tác vụ *remove, insert* vì lúc này chúng ta không phải dời nút mà chỉ sửa lại các vùng liên kết cho phù hợp. Thời gian thực hiện các tác vụ này không phụ thuộc vào số lượng các nút có trong danh sách liên kết.

Tuy nhiên, vì mỗi nút của danh sách liên kết phải chứa thêm trường next nên không sử dụng tối ưu bộ nhớ, việc truy xuất nút thứ i trên danh sách liên kết sẽ chậm vì phải truy xuất các phần tử một cách tuần tự bằng cách duyệt từ đầu danh sách, các tác vụ tìm kiếm trên danh sách liên kết cũng không tối ưu vì thường phải dùng phương pháp tìm kiếm tuyến tính.

2.4 HIỆN THỰC DANH SÁCH KỀ

2.4.1 Khai báo cấu trúc của danh sách kề

Là một mẩu tin có hai trường:

- Trường *num*: lưu số nút hiện có trong danh sách.

- Trường *nodes*: là mảng một chiều, mỗi phần tử của mảng là một nút của danh sách.

```
#define MAXLIST 100
typedef int DataType;
typedef struct list{
    int num;
  DataType nodes[MAXLIST];
}List;
```

Lưu ý:

- DataType là kiểu dữ liệu cơ bản (int, float, long, double,...) hoặc kiểu dữ liệu cấu trúc do người dùng định nghĩa.
- Khi khai báo kích thước mảng (MAXLIST) đủ lớn để có thể chứa hết các nút của danh sách kề.
- Khi danh sách bị rỗng thì không thể hiện thực tác vụ xóa một phần tử ra khỏi danh sách.
- Khi danh sách bị đầy thì không thể thực hiện tác vụ thêm vào.

2.4.2 Các tác vụ trên danh sách kề

- Khởi động danh sách:

```
void Init(List &plist){
    plist.num=0;
}
```

- Xác định số nút của danh sách:

```
int ListSize(List plist){
    return plist.num;
}
```

- Kiểm tra danh sách rỗng:

```
int IsEmpty(List plist){
    return (plist.num==0);
}
```

- Kiểm tra danh sách đầy:

```
int IsFull(List plist){
    return (plist.num==MAXLIST);
}
```

- Truy xuất một phần tử của danh sách:

```
int Retrieve(List plist, int pos)
{
    if(pos<0 || pos>=ListSize(plist))
    {
        printf("Vi tri %d khong hop le",pos);
        return 0;
    }
    else
            if(IsEmpty(plist))
            {
                    printf("danh sach rong");
                    return 0;
            }
            else
                    return plist.nodes[pos];
}
```

- Thêm một phần tử mới vào danh sách:

```
void Insert(List &plist, int pos, int x){
    int i;
    if(pos <0 || pos> ListSize(plist))
            printf("\n Vi tri chen khong phu hop");
    else{
            if(IsFull(plist)){
                    printf("Danh Sach bi day");
            return;
            }
        else{
                    for(i=ListSize(plist)-1; i>=pos; i--){
```

```
                    plist.nodes[i+1]=plist.nodes[i];
                }
                plist.nodes[pos]=x;
            plist.num++;
            }
    }
}
```

Trước khi thêm:

30	24	15	28	12	22	17			
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Sau khi thêm nút 18 vào vị trí 3:

30	24	15	**18**	28	12	22	17		
0	1	2	3	4	5	6	7	8	9

Hình 2.3: Thêm nút 18 vào vị trí 3 trong danh sách

=> Cách khác, có thể dùng hàm int Insert(List &plist, int pos, int x) trả về True nếu thêm được, False không thêm được.

- *Xoá một phần tử ra khỏi danh sách:*

```
int Remove(List &plist, int pos){
    int x = -1;
    if(pos <0 || pos>=ListSize(plist))
            return x;
    else{
            if(IsEmpty(plist)) return x;
            else{
                    x=plist.nodes[pos];
                    for(int i=pos;i<ListSize(plist)-1;i++){
                    plist.nodes[i]=plist.nodes[i+1];
                    }
                    plist.num--;
            }
    }
```

```
    return x;
}
```

Trước khi xóa:

30	24	**15**	28	12	22	17			
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Sau khi xóa nút 15 tại vị trí 2:

30	24	28	12	22	17				
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Hình 2.4. Xóa nút 15 ở vị trí 2 trong danh sách

- *Thay thế một phần tử của danh sách:*

```
void Replace(List &plist, int pos, int x){
    if(pos<0 || pos>=ListSize(plist)){
            printf("\n Vi tri hieu chinh khong dung");
            return;
    }else{
            if(IsEmpty(plist)){
                    printf("\n Danh sach khong co phan tu");
                    return;
            }else
                    plist.nodes[pos]=x;
    }
}
```

- *Duyệt danh sách:*

```
void ShowList(List plist){
    int i;
    if(IsEmpty(plist)){
                    printf("\n Danh sach khong co phan tu");
            return;
    }
    for(i=0;i<plist.num;i++){
    printf("\n%d", plist.nodes[i]);
    }
}
```

- *Tìm kiếm một phần tử trong danh sách:*

```
Int Search(List plist, int x){
    int vitri=0;
    while(plist.nodes[vitri]!=x && vitri<plist.num)
        vitri++;
    if(vitri==plist.num)
        return -1;
    return vitri;
}
```

- *Sắp xếp các phần tử bên trong danh sách:*

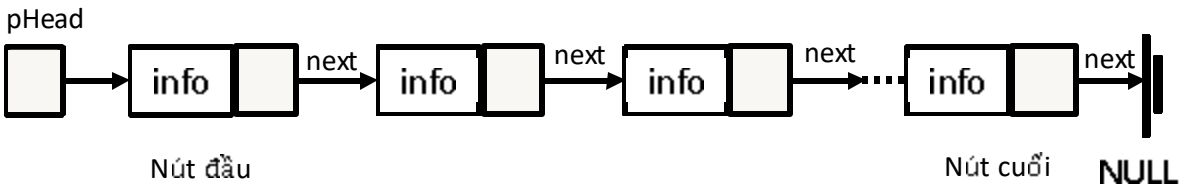
```
void SelectionSort(List &plist){
    int i,j,vitrimin,min;
    for(i=0;i<plist.num-1;i++){
            min=plist.nodes[i];
            vitrimin=i;
            for(j=i+1;j<plist.num;j++){
                    if(min >plist.nodes[j]){
                            min=plist.nodes[j];
                            vitrimin=j;
                    }
            }
            plist.nodes[vitrimin]=plist.nodes[i];
            plist.nodes[i]=min;
    }
}
```

2.5 HIỆN THỰC DANH SÁCH LIÊN KẾT ĐƠN

2.5.1 Giới thiệu danh sách liên kết đơn:

Danh sách liên kết đơn là một danh sách có nhiều nút và các nút của nó có thứ tự. Mỗi nút là một cấu trúc có trường *info* - chứa nội dung thật sự của nút và trường *next* là con trỏ chỉ nút tiếp theo trong danh sách liên kết. Thứ tự của các nút được thể hiện qua trường next: con trỏ đầu (*phead*) chỉ nút đầu tiên trong danh sách liên kết, nút đầu chỉ nút thứ hai, ..., nút cuối cùng của danh sách liên kết đơn là nút có trường next có giá trị NULL.

Hình vẽ sau đây mô tả một danh sách liên kết đơn:



Hình 2.5: Danh sách liên kết đơn

Lưu ý:

- Khi khởi động danh sách liên kết đơn, con trỏ đầu *phead* được gán giá trị NULL: *phead=NULL*; danh sách liên kết lúc này bị rỗng.
- Khi danh sách bị rỗng thì không thể thực hiện tác vụ xoá một phần tử, vì vậy trước khi thực hiện tác vụ xoá, chúng ta nên kiểm tra danh sách có bị rỗng hay không.
- Khi duyệt danh sách liên kết đơn nhờ con trỏ đầu *phead* chúng ta truy xuất được nút đầu và cứ lần theo liên kết có ở từng nút để tuần tự truy xuất đến nút cuối cùng của danh sách liên kết.

2.5.2 Khai báo cấu trúc danh sách liên kết đơn

Khai báo mỗi cấu trúc là một mẫu tin có hai trường *info* và trường *next*:

- Trường *info*: chứa nội dung của nút.
- Trường *next*: là con trỏ chỉ nút, dùng để chỉ đến nút kế tiếp trong danh sách liên kết.

```
typedef struct node{
    DataType info;
    struct node* next;
}Node;
typedef Node* NODEPTR;
```

2.5.3 Các tác vụ trên danh sách liên kết đơn

(Minh họa với *info* của nút kiểu *int*) Init, IsEmpty, InsertFirst, InsertAfter, DeleteFirst, DeleteAfter, DeleteAll, ShowList, Search, Sort

- *Tác vụ Init:* khởi động danh sách liên kết.

```
void Init(NODEPTR &phead){
    phead = NULL;
}
```

- *Tác vụ IsEmpty:* kiểm tra danh sách có rỗng hay không.

```
int IsEmpty (NODEPTR phead) {
    return (phead == NULL);
}
```

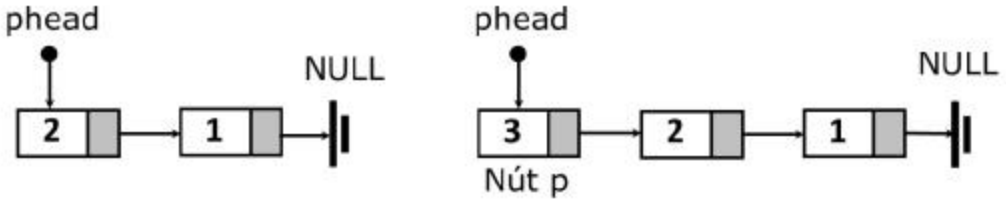
- *Tác vụ createNode:* cung cấp một biến động làm một nút cho danh sách liên kết - tạo nút có dữ liệu *x*.

```
NODEPTR CreateNode(DataType x){
    NODEPTR p = new Node;
    p->info = x;
    p->next = NULL;
    return p;
}
```

- *Tác vụ InsertFirst:* thêm một nút có nội dung x vào đầu danh sách liên kết.

```
void InsertFirst(NODEPTR &phead, DataType x){
    NODEPTR p = CreateNode(x);
    p->next = phead;
```

```
    phead = p;
}
```

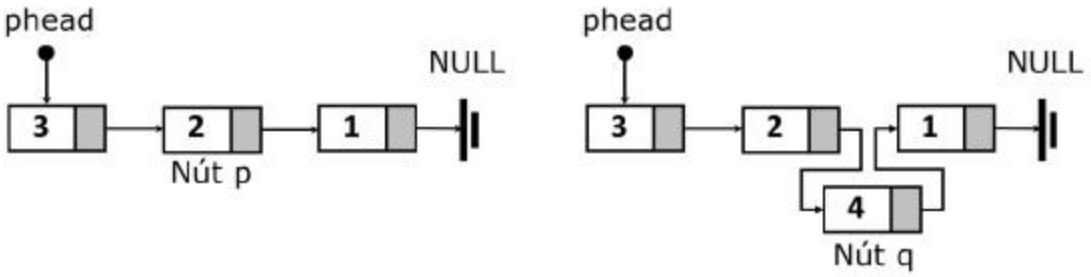


(a) (b)

Hình 2.6: Ví dụ minh họa tác vụ InsertFirst: (a) Danh sách liên kết ban đầu, (b) Danh sách liên kết sau khi thêm nút p vào đầu danh sách.

- *Tác vụ InsertAfter:* thêm một nút có nội dung x ngay sau nút p.

```
void InsertAfter (NODEPTR p, DataType x){
    NODEPTR tam;
    if(p!=NULL){
        q = createNode(x);
        q->next = p->next;
        p->next = q;
}
```



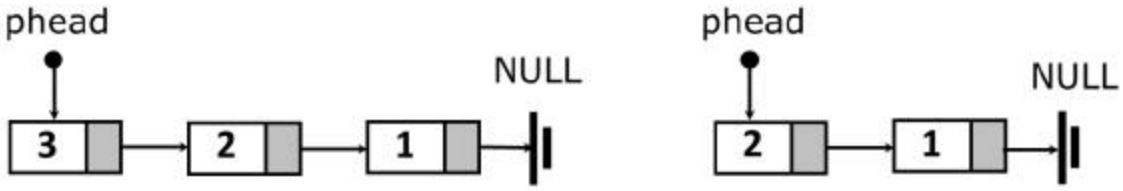
(a) (b)

Hình 2.7: Ví dụ minh họa tác vụ InsertAfter: (a) Danh sách liên kết ban đầu, (b) Danh sách liên kết sau khi thêm nút q vào ngay sau nút p.

- Tác vụ *DeleteFirst*: xóa nút đầu của danh sách liên kết.

```
void DeleteFirst(NODEPTR &pHead){
    NODEPTR p;
    if (IsEmpty(pHead))
        printf("List is empty!");
    else {
```

```
            p = pHead;
            pHead = pHead->next;
            delete p;
        }
    }
```

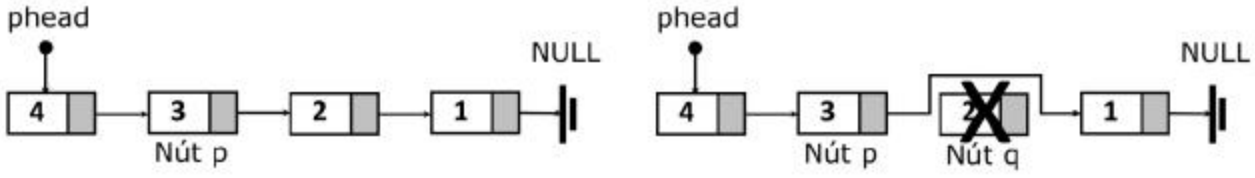


(a) (b)

Hình 2.8. Ví dụ minh họa tác vụ DeleteFirst: (a) Danh sách liên kết ban đầu, (b) Danh sách liên kết sau khi xóa nút đầu.

- Tác vụ *DeleteAfter*: xóa nút sau nút p trong danh sách

```
void DeleteAfter(NODEPTR p){
    NODEPTR q;
    if (p==NULL || p->next ==NULL) //kiem tra nut sau p co ton tai ko
            printf("Cannot delete node!");
    else    {
            q = p->next;
            p->next = q->next;
            delete q;
    }
}
```



(a) (b)

Hình 2.9. Ví dụ minh họa tác vụ DeleteAfter: (a) Danh sách liên kết ban đầu, (b) Danh sách liên kết sau khi xóa nút sau nút p.

- *Tác vụ ClearList:* xoá danh sách liên kết bằng cách giải phóng tất cả các nút có trên danh sách.

```
void ClearList(NODEPTR &phead){
    NODEPTR p;
```

```
    while (pHead!=NULL)
    {
            p = pHead;
            pHead = p->next;
            delete p;
    }
}
```

- *Tác vụ ShowList:* duyệt danh sách liên kết, hiện thị thông tin các nút.

```
void ShowList(NODEPTR phead){
    NODEPTR p = phead;
    if(p == NULL)
            printf("\n Danh sach bi rong");
    while(p != NULL){
            printf("%d",p->info);
            p = p->next;
    }
}
```

- *Tác vụ Search:* tìm kiếm nút có nội dung x trên danh sách liên kết bằng phương pháp tìm tuyến tính.

```
NODEPTR Search(NODEPTR phead, DataType x){
    NODEPTR p = phead;
    while(p->info !=x && p!=NULL)
            p = p->next;
    return p;
}
```

- *Tác vụ Sort:* sắp xếp danh sách liên kết theo giá trị tăng dần (dùng Selection Sort).

```
void Sort(NODEPTR &phead){
    NODEPTR p,q,pmin;
    int min;
    for(p = phead;p->next != NULL;p = p->next){
            min = p->info;
            pmin = p;
            for(q = p->next;q != NULL;q = q->next){
                    if(q->info < min){
```

```
                        min = q->info;
                        pmin = q;
                    }
                }
                pmin->info = p->info;
                p->info = min;
            }
        }
```

Một số tác vụ khác

- *Tác vụ nodePointer:* xác định nút thứ i trong danh sách liên kết, trả về địa chỉ của nút thứ i. phead->info

```
NODEPTR NodePointer(NODEPTR phead, int i){
    NODEPTR p=phead;
    int vitri=0;
    while(p!=NULL && vitri<i){
            p=p->next;
            vitri++;
    }
    return p;
}
```

- *Tác vụ Position:* xác định vị trí của nút p trong danh sách liên kết.

```
int Position(NODEPTR phead, NODEPTR p){
    int vitri=0;
    NODEPTR q=phead;
    while(q!=NULL && q!=p){
            q=q->next;
            vitri++;
    }
    if(q==NULL)
            return -1;
    return vitri;
}
```

- *Tác vụ prenode:* xác định nút trước của nút p trong danh sách liên kết.

```
NODEPTR PreNode(NODEPTR phead, NODEPTR p){
    NODEPTR q;
```

```
    if(p==phead)
        return NULL;
    q=phead;
    while(q!=NULL && q->next !=p)
        q=q->next;
    return q;
}
```

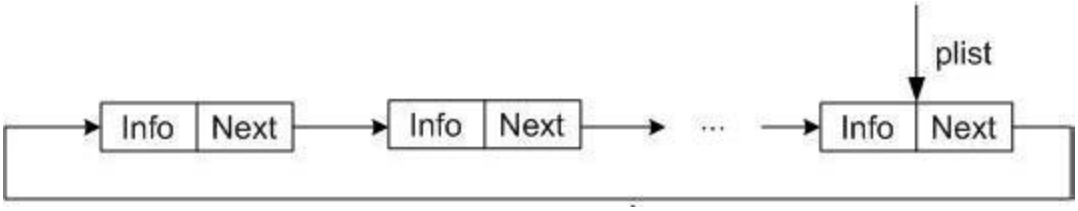
- *Tác vụ Place:* thêm một nút có nội dung x trên danh sách liên kết có thứ tự. Giả sử trường info của các nút có thứ tự tăng dần từ nhỏ đến lớn.

```
void Place(NODEPTR &phead, DataType x){
    NODEPTR p,q;
    q=NULL;
    for(p=phead; p!=NULL && x>p->info; p=p->next){
        q=p;
}
if(q==NULL)
    InsertFirst(phead,x);
else
    InsertAfter(q,x);
}
```

2.6 CÁC LOẠI DANH SÁCH LIÊN KẾT KHÁC

2.6.1 Danh sách liên kết vòng

Danh sách liên kết vòng là danh sách liên kết nhưng trường *next* của nút cuối chỉ nút đầu tiên của danh sách.



Hình 2.10: Danh sách liên kết vòng.

Lưu ý:

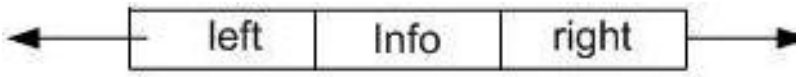
- Chúng ta quy ước *plist* trỏ đến nút cuối của danh sách liên kết vòng.
- Khi khởi động danh sách *plist* được gán bằng NULL.

- Với danh sách liên kết vòng khi biết con trỏ p của một nút chúng ta có thể truy xuất bất kỳ nút nào trong danh sách bằng cách lần theo vòng liên kết.

2.6.2 Danh sách liên kết kép

Danh sách liên kết kép là danh sách liên kết mà mỗi nút có hai trường liên kết: một trường liên kết chỉ nút trước (trường *left*) và một trường liên kết chỉ nút sau (trường *right*).

Hình ảnh sau mô tả một nút của danh sách liên kết kép.



Hình 2.11: Một nút của danh sách liên kết kép.

Hình ảnh sau mô tả một danh sách liên kết kép với plist là con trỏ chỉ nút đầu tiên của danh sách liên kết kép.

Hình 2.12: Danh sách liên kết kép

Với danh sách liên kết kép chúng ta có thể duyệt danh sách liên kết theo thứ tự xuôi danh sách (lần theo liên kết *right*) hoặc duyệt ngược danh sách (lần theo liên kết *left*). Nút cuối của danh sách liên kết có trường *right* chỉ NULL, nút đầu của danh sách liên kết có trường *left* chỉ NULL.

TÓM TẮT

Trong bài này, học viên cần nắm:

Cấu trúc dữ liệu danh sách là dãy các phần tử có cùng kiểu dữ liệu, và có tính thứ tự. Mỗi phần tử được lưu trong một nút

Hai cách cài đặt danh sách:

Cài đặt theo kiểu kế tiếp → hiện thực danh sách kề (dùng mảng một chiều)

Cài đặt theo kiểu liên kết → hiện thực danh sách liên kết đơn

Dạng mở rộng của danh sách liên kết đơn: danh sách liên kết vòng, danh sách liên kết kép, danh sách liên kết vòng kép.

CÂU HỎI ÔN TẬP

Câu 1: So sánh ưu khuyết điểm của danh sách liên kết đơn với danh sách kề.

Câu 2: Cài đặt các tác vụ bổ sung trên danh sách liên kết đơn

a. Thêm vào cuối danh sách

b. Sắp xếp danh sách theo phương pháp Interchange Sort

c. Xoá 1 phần tử có khoá là x

d. Thêm phần tử x vào ds đã có thứ tự (tăng) sao cho sau khi thêm vẫn có thứ tự (tăng).

e. Xác định vị trí của node x trong danh sách

f. Xác định kích thước của danh sách (số phần tử)

g. Chèn một phần tử có khoá x vào vị trí pos trong ds

h. Xoá các phần tử trùng nhau trong danh sách, chỉ giữ lại duy nhất một phần tử (*)

i. Trộn hai danh sách có thứ tự tăng thành một danh sách cũng có thứ tự tăng. (*)

Câu 3: Viết chương trình quản lý danh sách sinh viên (sử dụng DSLKĐ), thông tin mỗi sv gồm: Mã sv - chuỗi tối đa 10 kí tự, Họ tên - chuỗi tối đa 40 kí tự, Điểm trung bình - số thực. Chương trình có các chức năng sau:

a. Tạo 1 danh sách gồm n SV (n nhập từ bàn phím, thông tin của mỗi sv nhập từ bàn phím)

b. Xuất danh sách sinh viên

c. Xuất thông tin các sv có DTB>5

d. Tìm sinh viên có tên là X

Câu 4: Viết chương trình hiện thực danh sách liên kết kép (liên kết đôi).

Câu 5: Viết chương trình hiện thực danh sách liên kết vòng

Câu 6: Xây dựng cấu trúc danh sách liên kết kép vòng, mỗi nút trên danh sách có hai trường liên kết: Prev: trỏ đến nút trước, Next: trỏ đến nút sau, nút cuối cùng trong danh sách có trường next là nút đầu tiên, nút đầu tiên có trường prev là nút cuối cùng. Các thao tác trên danh sách:

Init, IsEmpty, CreateNode, InsertFrist, InsertLast, InsertPrev, InsertNext, InsertPos, DeleteFirst, DeleteLast, DeleteNext, DeletePrev, DeletePos, ShowList, ShowInvert, Search, Sort. ClearList.

BÀI 3: CẤU TRÚC STACK VÀ QUEUE

Sau khi học xong bài này, học viên có thể:

- *Hiểu được cấu trúc Stack và Queue.*
- *Cài đặt được các thao tác trên Stack và Queue.*
- *Vận dụng cấu trúc Stack và Queue để giải các bài toán cụ thể.*

3.1 GIỚI THIỆU VỀ *STACK*

3.1.1 Khái niệm về Stack

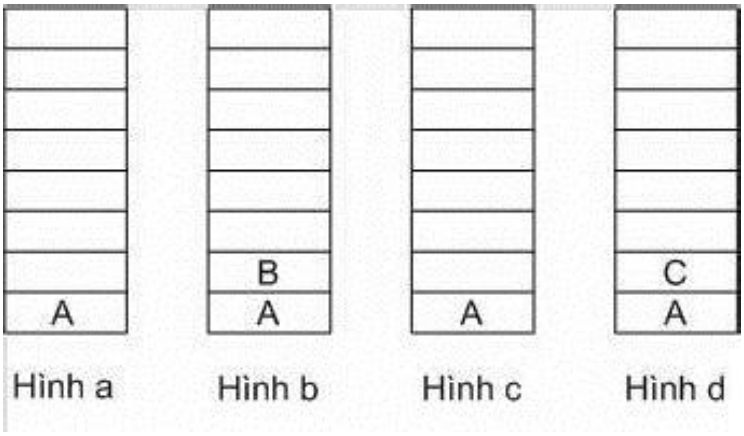
Stack có thể được xem là một dạng danh sách đặc biệt trong đó các tác vụ thêm vào hoặc xóa đi một phần tử chỉ diễn ra ở một đầu gọi là đỉnh *Stack*. Trên *Stack* các nút được thêm vào sau lại được lấy ra trước nên cấu trúc *Stack* hoạt động theo cơ chế vào sau ra trước - LIFO (Last In First Out).

Hai thao tác chính trên *Stack:*

- Tác vụ *push* dùng để thêm một phần tử vào đỉnh *Stack*
- Tác vụ *pop* dùng để xoá đi một phần tử ra khỏi đỉnh *Stack*.

Hình sau đây mô tả hình ảnh của *Stack* qua các tác vụ:

- push(s, A): hình a.
- push(s, B): hình b.
- pop(s): hình c.
- push(s, C): hình d.



Hình 3.1: *Stack* và thao tác push, pop trên nó.

3.1.2 Mô tả Stack

Mô tả dữ liệu:

Stack là một kiểu dữ liệu trừu tượng có nhiều nút cùng kiểu dữ liệu trải dài từ đáy *Stack* đến đỉnh *Stack*.

Mô tả các tác vụ:

- Tác vụ *Init*

  Chức năng: khởi động *Stack*

  Dữ liệu nhập: không

  Dữ liệu xuất: *Stack* top trở về đầu *Stack*.

- Tác vụ *IsEmpty*

  Chức năng: kiểm tra *Stack* có rỗng hay không.

  Dữ liệu nhập: không.

  Dữ liệu xuất: TRUE|FALSE.

- Tác vụ *Push*

  Chức năng: thêm nút mới tại đỉnh *Stack*.

  Dữ liệu nhập: nút mới

  Dữ liệu xuất: không.

- Tác vụ *Pop*

  Chức năng: xóa nút tại đỉnh *Stack*.

  Dữ liệu nhập: không

  Điều kiện: *Stack* không bị rỗng.

  Dữ liệu xuất: nút bị xoá.

3.1.3 Ứng dụng của Stack

- *Stack* thường được dùng để giải quyết các vấn đề có cơ chế LIFO.
- *Stack* thường được dùng để giải quyết các vấn đề trong trình biên dịch của các ngôn ngữ lập trình như:
  - Kiểm tra cú pháp của các câu lệnh trong ngôn ngữ lập trình.
  - Xử lý các biểu thức toán học: kiểm tra tính hợp lệ của các dấu trong ngoặc một biểu thức, chuyển biểu thức từ dạng trung tố (infix) sang dạng hậu tố (postfix), tính giá trị của biểu thứ dạng hậu tố.
  - Xử lý việc gọi các chương trình con.
- *Stack* thường được sử dụng để chuyển một giải thuật đệ qui thành giải thuật không đệ qui.

3.2 HIỆN THỰC *STACK*

3.2.1 Hiện thực Stack bằng danh sách kề

Khai báo cấu trúc *Stack*: là môt mẩu tin có hai trường:

- Trường top: là một số nguyên chỉ đỉnh *Stack*.
- Trường nodes: là mảng một chiều, mỗi phần tử của mảng là một nút của *Stack*.

```
#define MAXSTACK 100
struct stack{
    int top;
    int nodes[MAXSTACK];
};
```

```
typedef struct stack Stack;
```

Các tác vụ trên Stack

- Khởi tạo Stack

```
void Init(Stack &s){
    s.top = -1;
}
```

- Tác vụ IsEmpty – kiểm tra Stack rỗng

```
int IsEmpty(Stack s){
    return (s.top==-1);
}
```

- Tác vụ IsFull – kiểm tra Stack đầy

```
int IsFull(Stack s){
    return (s.top == MAXSTACK-1);
}
```

- Tác vụ Push

```
void Push(Stack  &s, int x){
    if(IsFull(s)) return 0;
     else{
        s.nodes[++s.top]=x;
        return 1;
     }
}
```

- Tác vụ Pop

```
int Pop(Stack &s){
        if(IsEmpty(s))  return 0;
    else{
        x = s.nodes[s.top--];
        return 1;
    }
}
```

3.2.2 Hiện thực stack bằng danh sách liên kết

Khai báo cấu trúc Stack:

```
typedef struct node
{
    DataType      info;
    struct node * next;
}Node;
typedef Node* NODEPTR;
typedef  NODEPTR STACk;
STACK s;
```

Các tác vụ trên Stack:

- *Tác vụ khởi tạo Stack*

```
void Init(STACK &s){
    s=NULL;
}
```

- *Tác vụ kiểm tra Stack rỗng*

```
int IsEmpty(STACK s) {
    return (s==NULL);
}
```

- *Tác vụ thêm một phần tử vào Stack*

```
int Push(STACK &s, DataType x){
    NODEPTR p = new Node;
  if(p==NULL) return 0;
    p->info = x;
    p->next = s;
    s = p;
  return 1;
}
```

- *Tác vụ lấy một phần tử ra khỏi Stack*

```
int Pop(STACK &s, DataType &x)
```

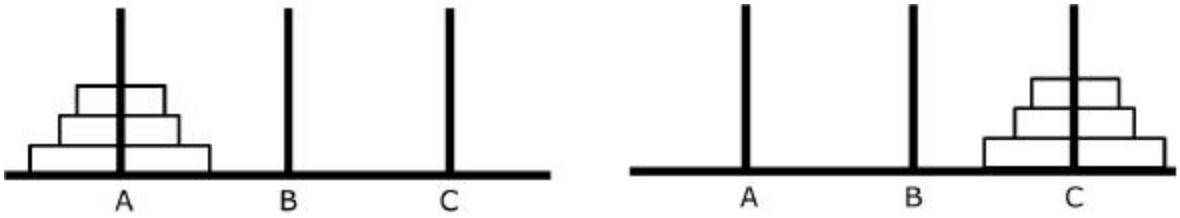
```
{
    if (isEmpty(s))        return 0;
    NODEPTR p=s;
    x=p->info;
    s=s->next;
  delete p;
  return 1;
}
```

3.3 MỘT SỐ BÀI TOÁN ỨNG DỤNG STACK

- Bài toán tháp Hà Nội: Khử đệ quy
- Áp dụng cho bài toán dùng cơ chế LIFO: Chuyển biểu thức trung tố (Infix) sang biểu thức hậu tố (Postfix) hoặc biểu thức tiền tố (Prefix); Tính giá trị biểu thức hậu tố.

3.3.1 Bài toán tháp Hà Nội: Khử đệ quy

Bài toán tháp Hà Nội được phát biểu như sau: "Cho ba cọc A, B, C và *n* đĩa có kích thước khác nhau có thể cho vào các cọc này. Ban đầu sắp xếp các đĩa theo trật tự kích thước vào cọc A sao cho đĩa nhỏ nhất nằm trên cùng. Người chơi phải di chuyển toàn bộ số đĩa sang cọc C khác, tuân theo các quy tắc sau: (1) một lần chỉ được di chuyển một đĩa, (2) một đĩa chỉ có thể được đặt lên một đĩa lớn hơn".



(a) (b)

Hình 3.2: Bài toán tháp Hà Nội: (a) Trạng thái bắt đầu, (b) Trạng thái kết thúc.

Bài toán tháp Hà Nội có thể được giải bằng kỹ thuật đệ quy như sau:

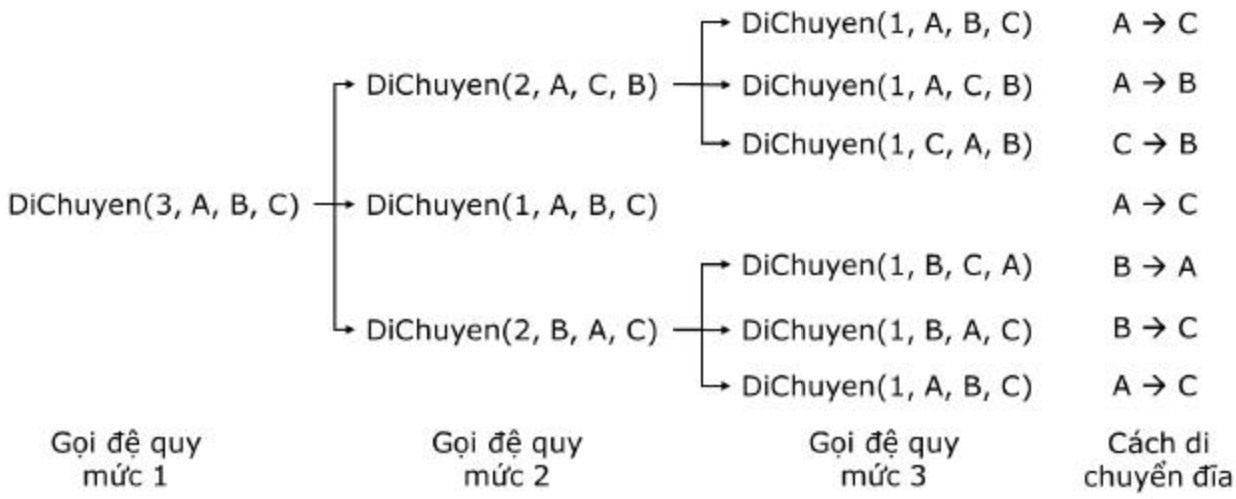
Bước 1: Chuyển n-1 đĩa từ A sang B

Bước 2: Chuyển đĩa cuối cùng từ A sang C

Bước 3: Chuyển n-1 đĩa từ B sang C

```
void DiChuyen(int n , char a, char b, char c )
{
    if (n==1)
   {
        printf("Di chuyen 1 dia tu %c sang %c\n", a, c);
        return;
    }
    DiChuyen (n-1, a, c, b);      // Bước 1
    DiChuyen (1, a, b, c);// Bước 2
    DiChuyen (n-1, b, a, c);      // Bước 3
}
```

Giải thuật đệ quy trên được minh họa với n = 3 trong hình sau đây:



Hình 3.3: Cây đệ quy giải bài toán tháp Hà Nội với số lượng đĩa bằng 3.

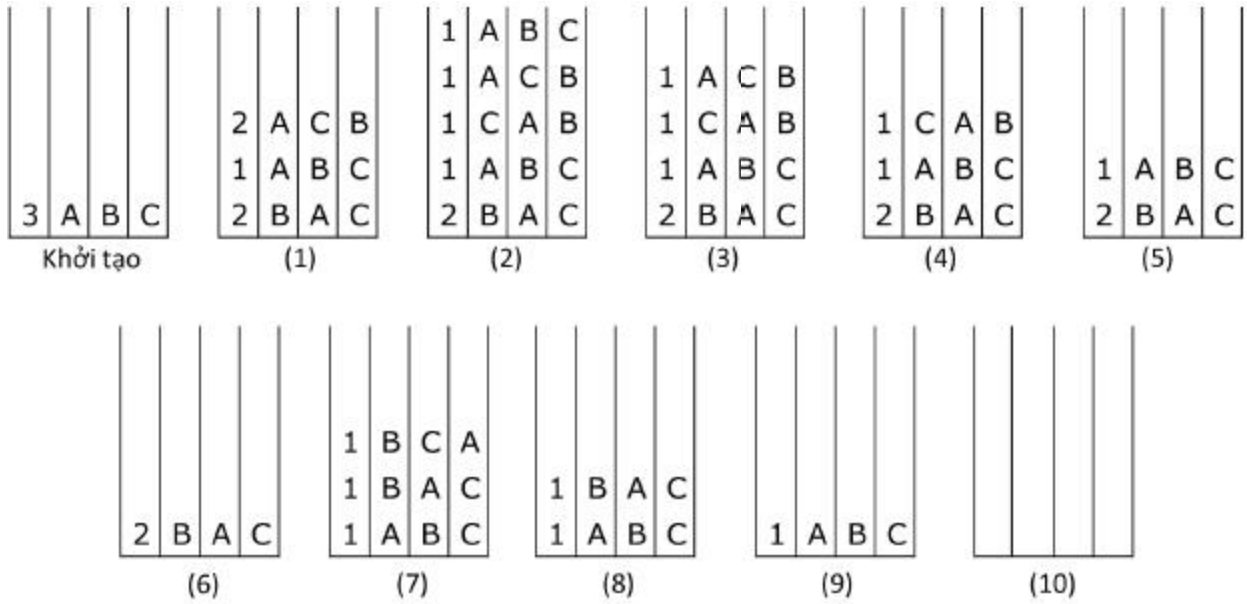
Tuy giải thuật đệ quy đã giải quyết bài toán tháp Hà Nội một cách có hiệu quả. Trong một số trường hợp, chẳng hạn để tối ưu bộ nhớ và thời gian, chúng ta có thể thực hiện khử đệ quy giải thuật giải bài toán tháp Hà Nội nói trên. Để khử đệ quy, chúng ta sẽ sử dụng ngăn xếp để mô phỏng lại cây đệ quy. Giải thuật như sau:

```
struct item_tower
{
    int num;
    char A, B, C;
    void assign(int n, char a, char b, char c)
```

```
    {
        num = n;
        A = a;
        B = b;
        C = c;
    }
};
void DiChuyen_KhuDeQuy(int n, char a, char b, char c)
{
    stack s;
    item t;
    int d = 0;
    t.assign(n, a, b, c);

    init(s);
    push(t, s);

    while(isEmpty(s) == false)
    {
        item temp = top(s);
        pop(s);
        if (temp.num == 1)
            printf("Di chuyen 1 dia tu %c sang %c\n ", temp.A, temp.C);
        else
        {
            t.assign(temp.num-1, temp.B, temp.A, temp.C);
            push(t, s);

            t.assign(1, temp.A, temp.B, temp.C);
            push(t, s);

            t.assign(temp.num-1, temp.A, temp.C, temp.B);
            push(t, s);
        }
    }
}
```

Hình 3.4 minh họa sự thay đổi của ngăn xếp s trong hàm DiChuyen_KhuDeQuy theo từng lần lặp. Tại lần lặp thứ 3, ta xác định được bước di chuyển đĩa đầu tiên từ cột A sang cột C. Tại lần lặp thứ 4, xác định được bước di chuyển đĩa từ cột A sang B. Tại lần lặp thứ 5, xác định được bước di chuyển đĩa từ cột C sang B. Tại lần lặp thứ 6,

xác định được bước di chuyển đĩa từ cột A sang C. Tại lần lặp thứ 8, xác định được bước di chuyển đĩa từ cột B sang A. Tại lần lặp thứ 9, xác định được bước di chuyển đĩa từ cột B sang C. Tại lần lặp thứ 10, xác định được bước di chuyển đĩa từ cột A sang C. Sau 10 lần lặp, hàm dừng do ngăn xếp không còn phần tử.



Hình 3.4: Minh họa sự thay đổi của ngăn xếp s với số lượng đĩa bằng 3

3.3.2 Bài toán chuyển đổi dạng của biểu thức

Một biểu thức toán học có thể được biểu diễn dưới dạng biểu thức trung tố (infix), biểu thức hậu tố (postfix hay ký pháp Ba Lan ngược), hoặc biểu thức tiền tố (prefix hay ký pháp Ba Lan).

Biểu thức trung tố là cách biểu diễn biểu thức toán học thông dụng nhất mà chúng ta đã biết. Biểu thức A × (B + C) / D là một ví dụ của biểu thức trung tố. Biểu thức trung tố có các đặc điểm sau: (1) Toán tử (phép toán) được viết giữa các toán hạng, (2) Các phép toán có thứ tự ưu tiên, (3) Thứ tự thực hiện các phép toán từ trái qua phải nếu các phép toán có cùng thứ tự ưu tiên, (4) Sử dụng dấu ngoặc đơn () để thay đổi thứ tự thực hiện của phép toán.

Biểu thức hậu tố còn được gọi là ký pháp Ba Lan ngược. Trong biểu thức hậu tố, các toán tử (phép toán) được viết sau các toán hạng. Ngoài ra, biểu thức hậu tố còn có các đặc điểm sau: (1) Thứ tự thực hiện phép toán từ trái qua phải, (2) Không sử dụng dấu ngoặc đơn () để thay đổi thứ tự thực hiện của phép toán, (3) Phép toán được thực hiện trên các toán hạng nằm ngay trước phép toán.

Biểu thức tiền tố còn được gọi là ký pháp Ba Lan. Trong biểu thức tiền tố, các toán tử (phép toán) được viết trước các toán hạng. Ngoài ra, biểu thức tiền tố còn có các đặc điểm sau: (1) Thứ tự thực hiện phép toán từ trái qua phải, (2) Không sử dụng dấu ngoặc đơn () để thay đổi thứ tự thực hiện của phép toán, (3) Phép toán được thực hiện trên các toán hạng nằm ngay sau phép toán. Do trong biểu thức hậu tố và biểu thức tiền tố, các phép toán được thực hiện từ trái qua phải mà không cần bận tâm tới thứ tự ưu tiên của các phép toán, các biểu thức này giúp làm giảm độ phức tạp về bộ nhớ của thuật toán tính giá trị của biểu thức.

Ví dụ về biểu thức trung tố, tiền tố, và hậu tố được thể hiện ở bảng bên dưới. Trên mỗi dòng là các cách biểu diễn khác nhau của cùng một biểu thức.

Biểu thức trung tố	Biểu thức tiền tố	Biểu thức hậu tố
A × B + C / D	+ × A B / C D	A B × C D / +
A × (B + C) / D	/ × A + B C D	A B C + × D /
A × (B + C / D)	× A + B / C D	A B C D / + ×

Thuật toán chuyển biểu thức trung tố sang biểu thức hậu tố gồm các bước như sau:

1. Khởi tạo:

   Biến output = [] dùng để chứa biểu thức kết quả

   Stack s

2. Duyệt biểu thức từ trái qua phải.
3. Nếu ký tự đang xét là toán hạng, thêm ký tự đó vào ouput.
4. Nếu ký tự đang xét là '(', thêm ký tự đó vào stack s.
5. Nếu ký tự đang xét là ')', lấy một phần tử ra khỏi stack s và thêm phần tử đó vào output cho đến khi gặp dấu '('. Loại bỏ dấu '(' vừa gặp khỏi stack.
6. Nếu ký tự đang xét là toán tử:

   6.1. Lấy phần tử đầu ra khỏi stack s và thêm phần tử đó vào output nếu phần tử ở đỉnh stack là toán tử và toán tử đó có độ ưu tiên ***lớn hơn hoặc bằng*** toán tử đang xét.

6.2. Thêm ký tự đang xét vào stack s.

7. Lặp lại bước 3-6 cho tới khi duyệt tới phần tử cuối cùng của biểu thức.
8. Sau khi duyệt hết biểu thức, nếu trong stack còn phần tử thì thấy lần lượt các phần tử ra và thêm vào output.

Ví dụ 3.1: Chuyển biểu thức exp = A*(B+C) sang dạng hậu tố.

Khởi tạo: output = []

stack s = NULL

i	exp[i]	output	s
0	A	A	NULL
1	*	A	*
2	(	A	(*
3	B	AB	(*
4	+	AB	+(*
5	C	ABC	+(*
6	)	ABC+	*

Biểu thức hậu tố tương ứng là: **ABC+***

Ví dụ 3.2: Chuyển biểu thức exp = A*B^C+D sang dạng hậu tố

Khởi tạo: output = []

stack s = NULL

i	exp[i]	output	s
0	A	A	NULL
1	*	A	*
2	B	AB	*
3	^	AB	^*
4	C	ABC	^*
5	+	ABC^*	+
6	D	ABC^*D	+

Biểu thức hậu tố tương ứng là: **ABC^*D+**

Thuật toán chuyển biểu thức trung tố sang biểu thức tiền tố gồm các bước như sau:

1. Đảo ngược biểu thức trung tố.
2. Thay thế '(' bằng ')' và ')' bằng '(' trong biểu thức đảo ngược.
3. Thực hiện các bước tìm biểu thức hậu tố của biểu thức đảo ngược thu được ở bước 2 như sau:

   3.1. Khởi tạo:

   - Biến output = [] dùng để chứa biểu thức kết quả
   - Stack s

   3.2. Duyệt biểu thức từ trái qua phải.

   3.3. Nếu ký tự đang xét là toán hạng, thêm ký tự đó vào ouput.

   3.4. Nếu ký tự đang xét là '(', thêm ký tự đó vào stack s.

   3.5. Nếu ký tự đang xét là ')', lấy một phần tử ra khỏi stack s và thêm phần tử đó vào output cho đến khi gặp dấu '('. Loại bỏ dấu '(' vừa gặp khỏi stack.

   3.6. Nếu ký tự đang xét là toán tử:

      3.6.1. Lấy phần tử đầu ra khỏi stack s và thêm phần tử đó vào output nếu phần tử ở đỉnh stack là toán tử và toán tử đó có độ ưu tiên ***lớn hơn*** toán tử đang xét.

      3.6.2. Thêm ký tự đang xét vào stack s.

   3.7. Lặp lại bước 3.3-3.6 cho tới khi duyệt tới phần tử cuối cùng của biểu thức.

   3.8. Sau khi duyệt hết biểu thức, nếu trong stack còn phần tử thì thấy lần lượt các phần tử ra và thêm vào output.

4. Đảo ngược biểu thức hậu tố thu được ở bước 3.
5. Kết quả của bước 4 là biểu thức tiền tố cần tìm.

Ví dụ 3.3: Chuyển biểu thức exp = A*(B+C) sang dạng tiền tố.

Bước 1: rev =)C+B(*A

Bước 2: rev = (C+B)*A

i	exp[i]	output	S
0	(	[]	(
1	C	C	(
2	+	C	+(
3	B	CB	+(
4	)	CB+	
5	*	CB+	*
6	A	CB+A	*

Bước 3: thu được biểu thức hậu tố: CB+A*

Bước 4: biểu thức tiền tố cần tìm = ***A+BC**

Ví dụ 3.4: Chuyển biểu thức exp = A*B+C/D sang dạng tiền tố

Bước 1: rev = D/C+B*A

Bước 2: rev = D/C+B*A

i	exp[i]	output	S
0	D	D	NULL
1	/	D	/
2	C	DC	/
3	+	DC/	+
4	B	DC/B	+
5	*	DC/B	*+
6	A	DC/BA	*+

Bước 3: thu được biểu thức hậu tố: DC/BA*+

Bước 4: biểu thức tiền tố cần tìm **= +*AB/CD**

3.3.3 Bài toán tính giá trị biểu thức hậu tố

Thuật toán tính giá trị của biểu thức hậu tố gồm các bước như sau:

1. Khởi tạo stack s = NULL.
2. Duyệt biểu thức hậu tố từ trái qua phải.

3. Nếu phần tử đang xét là toán hạng, thêm phần tử đó vào stack s.
4. Nếu phần tử đang xét là toán tử, lấy phần tử đầu tiên (a) và phần tử thứ 2 (b) ra khỏi stack và thực hiện phép toán giữa b và a (kq). Sau đó thêm kết quả của phép toán vào stack s.
5. Lặp lại bước 3 và 4 cho tới khi duyệt tới phần tử cuối cùng của biểu thức.
6. Kết quả của biểu thức hậu tố chính là phần tử đầu tiên trong stack s thu được sau bước 5.

Ví dụ 3.5: Tính giá trị biểu thức 2578*+*9-

Khởi tạo stack s = NULL

i	exp[i]	a	b	Kết quả phép toán giữa b và a	S
0	2	-	-	-	2
1	5	-	-	-	5 2
2	7	-	-	-	7 5 2
3	8	-	-	-	8 7 5 2
4	*	8	7	7 * 8 = 56	56 5 2
5	+	56	5	5 + 56 = 61	61 2
6	*	61	2	2 * 61 = 122	122
7	9	-	-	-	9 122
8	-	9	122	122 – 9	**113**

Giá trị biểu thức 2578*+*9- là 113.

3.3.4 Bài toán tính giá trị biểu thức tiền tố

Thuật toán tính giá trị của biểu thức tiền tố gồm các bước như sau:

1. Khởi tạo stack s = NULL.
2. Duyệt biểu thức tiền tố từ phải qua trái.
3. Nếu phần tử đang xét là toán hạng, thêm phần tử đó vào stack s.
4. Nếu phần tử đang xét là toán tử, lấy phần tử đầu tiên (a) và phần tử thứ 2 (b) ra khỏi stack và thực hiện phép toán giữa a và b. Sau đó thêm kết quả của phép toán vào stack s.

5. Lặp lại bước 3 và 4 cho tới khi duyệt tới phần tử đầu tiên của biểu thức.
6. Kết quả của biểu thức tiền tố chính là phần tử đầu tiên trong stack s thu được sau bước 5.

Ví dụ 3.6: Tính giá trị biểu thức -*2+5*789

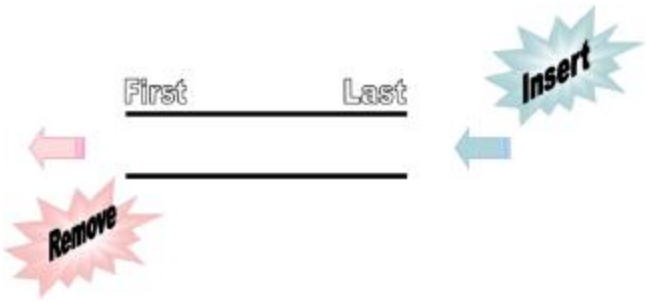
Khởi tạo stack s = NULL

i	exp[i]	a	b	Kết quả phép toán giữa a và b	S
8	9	-	-	-	9
7	8	-	-	-	8 9
6	7	-	-	-	7 8 9
5	*	7	8	7 * 8 = 56	56 9
4	5	-	-	-	5 56 9
3	+	5	56	5 + 56 = 61	61 9
2	2	-	-	-	2 61 9
1	*	2	61	2 * 61	122 9
0	-	122	9	122 – 9 = 113	**113**

Giá trị biểu thức -*2+5*789 là 113.

3.4 GIỚI THIỆU VỀ QUEUE

Queue (hàng đợi) là một dạng danh sách đặt biệt, trong đó chúng ta chỉ được phép thêm các phần tử vào cuối hàng đợi và lấy ra các phần tử ở đầu hàng đợi. Vì phần tử thêm vào trước được lấy ra trước nên cấu trúc hàng đợi còn được gọi là cấu trúc FIFO (First In First Out).



Hình 3.5: Cơ chế FIFO

Hàng đợi là cấu trúc được sử dụng rộng rãi trong thực tế: người ta dùng hàng đợi để giải quyết các vấn đề có cấu trúc FIFO như xử lý các dịch vụ của ngân hàng, để xử

lý các tiến trình đang đợi phục vụ trong các hệ điều hành nhiều người sử dụng, quản lý các yêu cầu in trên máy print servers...

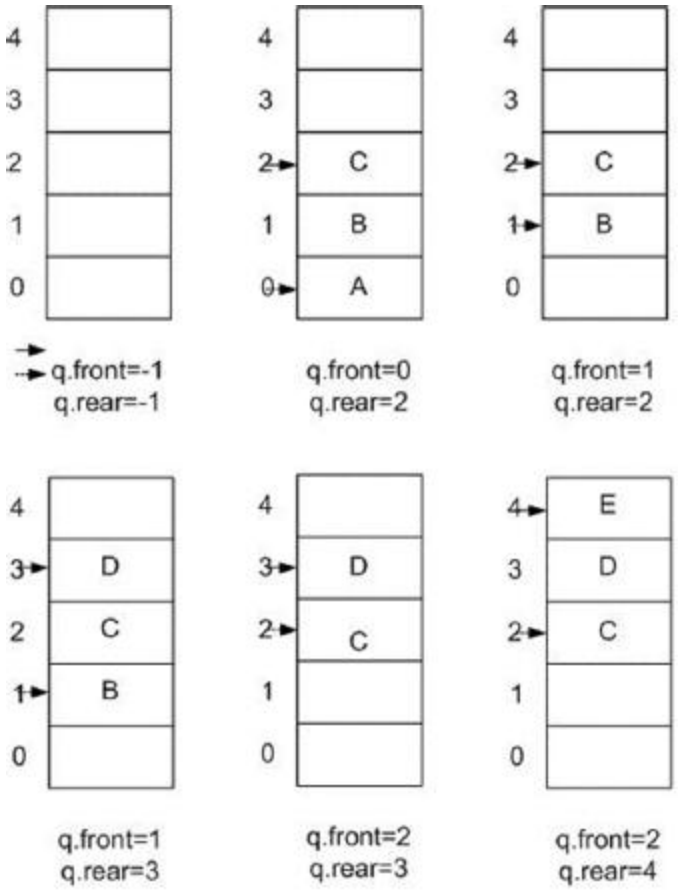
Khái niệm về Queue

Hàng đợi chứa các nút có cùng kiểu dữ liệu trải dài từ đầu hàng đợi (front) đến cuối hàng đợi (rear). Hai tác vụ chính trên hàng đợi:

- insert – thêm nút mới vào cuối hàng đợi
- remove – dùng để xoá một phần tử ra khỏi hàng đợi.

Hình vẽ sau đây dùng để mô tả hàng đợi qua các tác vụ như sau:

- Trạng thái bắt đầu: hàng đợi bị rỗng (hình a)
- insert (q, A)
- insert (q, B)
- insert (q, C) (hình b)
- remove(q) (hình c)
- insert (q, D) (hình d)
- remove(q) (hình e)
- insert (q, E) (hình f)



Hình 3.6: Minh hoa tác vụ insert và remove trên Queue

Mô tả Queue

Mô tả dữ liệu

Hàng đợi là một kiểu dữ liệu trừu tượng có nhiều nút cùng kiểu dữ liệu trải dài từ đầu hàng đợi đến cuối hàng đợi.

Mô tả các tác vụ cơ bản:

- Tác vụ *Init*

  Chức năng: khởi động hàng đợi

  Dữ liệu nhập: không

  Dữ liệu xuất: hai con trỏ front và rear được gán các giá trị phù hợp.

- Tác vụ *IsEmpty*

  Chức năng: kiểm tra hàng đợi có bị rỗng hay không

  Dữ liệu nhập: không

  Dữ liệu xuất: TRUE|FALSE

- Tác vụ *Insert*

  Chức năng: Thêm nút mới vào hàng đợi.

  Dữ liệu nhập: nút mới

  Điều kiện: hàng đợi không bị đầy.

  Dữ liệu xuất: không.

- Tác vụ *Remove*

  Chức năng: lấy nút tại đầu hàng đợi

  Dữ liệu nhập: không

  Điều kiện: hàng đợi không bị rỗng

  Dữ liệu xuất: thông tin nút bị lấy ra.

Ứng dụng của hàng đợi

Hàng đợi được dùng để giải quyết các vấn đề có cơ chế FIFO (First In First Out).

Trong thực tế người ta thường cài đặt hàng đợi trong các chương trình xử lý các dịch vụ ngân hàng, xử lý các tác vụ đang đợi phục vụ trong các hệ điều hành đa người sử dụng, quản lý các yêu cầu in trong máy server printers...

3.5 HIỆN THỰC QUEUE

3.5.1 Dùng mảng vòng hiện thực hàng đợi

Đây là cách thường dùng nhất để cài đặt hàng đợi theo kiểu kế tiếp. Lúc này ta xem mảng như là mảng vòng chứ không phải là mảng thẳng: nút node[0] xem như là nút sau của nút nodes[MAXQUEUE -1].

- *Khai báo cấu trúc:*

```
#define MAXQUEUE 100
typedef int DataType;
struct queue{
    int front, rear;
    DataType node[MAXQUEUE];
} Queue;
```

- *Tác vụ khởi động:*

```
void Init(Queue &q){
    q.front=q.rear= -1;
}
```

- *Tác vụ kiểm tra hàng đợi trống:*

```
int IsEmpty(Queue &pq){
    return (q.front==-1);
}
```

- *Tác vụ kiểm tra hàng đợi đầy:*

```
int IsFull(Queue &pq){
    return ((q.rear - q.front+1) % MAXQUEUE == 0 );
}
```

- *Tác vụ thêm vào hàng đợi:*

```
int enQueue (Queue &q, DataType x){
    if(isFull(q)) return 0; // Full Queue
    else
    {       if(isEmpty(q)) q.front = 0;
            q.rear = (q.rear+1) % MAXQUEUE;
            q.node[q.rear] = x;
            return 1;
    }
}
```

- *Tác vụ lấy một phần tử ra khỏi hàng đợi:*

```
int deQueue(Queue &q, DataType &x)
{
    if(isEmpty(q)) return 0; // Empty Queue
    else
    {       x = q.node[q.front];
            if(q.front == q.rear) init(q);
            else    q.front = (q.front+1) % MAXQUEUE;
            return 1;
    }
}
```

3.5.2 Dùng danh sách liên kết hiện thực hàng đợi

Ta có thể cài đặt hàng đợi như một danh sách liên kết: con trỏ đầu danh sách liên kết *front* chỉ đến nút tại đầu hàng đợi, con trỏ *rear* trỏ đến nút cuối cùng của hàng đợi.

Hình vẽ sau thể hiện cách cài đặt hàng đợi bằng danh sách liên kết.

Hình 3.7: Cài đặt Queue bằng danh sách liên kết

Với cách cài đặt này ta có thể hiện thực các tác vụ của hàng đợi như: Insert, Remove, IsEmpty…trên danh sách liên kết.

3.6 HÀNG ĐỢI CÓ ƯU TIÊN

Hàng đợi hoạt động trên nguyên tắc nút nào vào trước được lấy ra trước, nút nào vào sau được lấy ra sau là hàng đợi không có ưu tiên.

Trong thực tế có một dạng hàng đợi khác hoạt động theo cơ chế: nút nào có độ ưu tiên cao hơn sẽ được lấy ra trước, nút nào có độ ưu tiên thấp hơn thì lấy ra sau. Hàng đợi lúc này gọi là hàng đợi có ưu tiên (priority queue).

Có 2 loại hàng đợi có ưu tiên:

- Hàng đợi có ưu tiên tăng: nút có độ ưu tiên cao nhất được lấy ra.
- Hàng đợi có ưu tiên giảm: nút có độ ưu tiên giảm sẽ được lấy ra trước.

Phần hiện thực hàng đợi ưu tiên sẽ được xem như bài tập.

TÓM TẮT

Trong bài này, học viên cần nắm:

Cấu trúc Stack là danh sách đặc biệt với thao tác thêm vào và lấy ra chỉ được thực hiện ở một đầu của danh sách. Do đó, Stack hoạt động theo cơ chế LIFO (vào trước, ra sau). Ứng dụng của Stack: khử đệ quy, sử dụng trong các bài toán có cơ chế LIFO.

Các thao tác cơ bản trên Stack: Init, IsEmpty, Push, Pop.

Có thể dùng mảng, danh sách liên kết để mô tả danh sách các phần tử của Stack.

Cấu trúc Queue là danh sách đặc biệt với thao tác thêm vào được thực hiện ở cuối danh sách và lấy ra ở đầu danh sách. Queue hoạt động theo cơ chế FIFO.

Các thao tác cơ bản trên Queue: Init, isEmpty, Insert, Remove

Ứng dụng Queue trong bài toán hàng đợi "Vào trước ra trước" FIFO: hệ thống print server, cơ chế thông điệp, bộ đệm, hàng đợi xử lý sự kiện... Các ứng dụng đặt vé tàu lửa, máy bay... Các hệ thống rút tiền.

CÂU HỎI ÔN TẬP

Câu 1: Hiện thực *Stack* và các tác vụ của *Stack* bằng danh sách liên kết.

Câu 2: Viết chương trình đổi một số thập phân sang cơ số bất kỳ vận dụng *Stack*.

Câu 3: Viết chương trình cài đặt bài toán chuyển biểu thức trung tố sang hậu tố, sau đó tính giá trị biểu thức hậu tố.

Câu 4: Hiện thực hàng đợi và các tác vụ của hàng đợi bằng danh sách liên kết.

Câu 5: Viết chương trình quản lý kho hàng dùng hàng đợi được hiện thực bằng danh sách liên kết.

Câu 6: Viết chương trình hiện thực hàng đợi có độ ưu tiên.

BÀI 4: CẤU TRÚC CÂY - CÂY NHỊ PHÂN – CÂY NHỊ PHÂN TÌM KIẾM

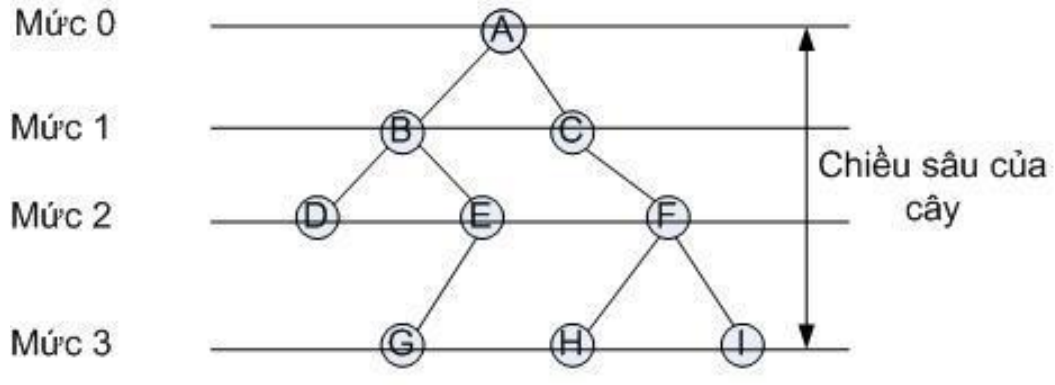
Sau khi học xong bài này, học viên có thể:

- *Hiểu được cấu trúc cây tổng quát, cấu trúc cây nhị phân, và cấu trúc cây nhị phân tìm kiếm.*
- *Cài đặt được các thao tác trên cây nhị phân và cây nhị phân tìm kiếm.*
- *Vận dụng cấu trúc cây nhị phân và cây nhị phân tìm kiếm để giải các bài toán cụ thể.*

4.1 CẤU TRÚC CÂY TỔNG QUÁT

Cây là một cấu trúc gồm một tập hữu hạn các nút cùng kiểu dữ liệu (tập nút này có thể rỗng), các nút được nối với nhau bởi cạnh. Có duy nhất một nút gốc, có duy nhất một đường đi từ nút gốc đến một nút. Các loại cây: cây nhị phân (mỗi nút có tối đa 2 nút con), cây tam phân, cây n-phân (mỗi nút có tối đa n nút con).

Các khái niệm cơ bản về cây



Hình 4.1: Minh họa các khái niệm về cây.

Nút gốc (root): là nút đầu tiên của cây, là nút không có nút cha, hình vẽ trên có A là nút gốc.

Nút lá (leaf): là nút không có con, ví dụ các nút D, G, H và I là các nút lá.

Nút trung gian hay nút trong (internal node): Nút trung gian là nút ở giữa cây nhị phân, nó không là nút lá cũng không phải là nút gốc. Ví dụ các nút B, C, E và F là những nút trung gian.

Nút anh em (brothers): Hai nút gọi là anh em với nhau nếu chúng có cùng một nút cha.

Bậc của nút (degree of node): Bậc của nút là số nút con của nút đó. Với cây nhị phân bậc của nút có 1 trong ba giá trị: 0, 1, 2. Ví dụ nút A có bậc của nút là 2, nút E có bậc của nút là 1, nút D có bậc của nút là 0.

Bậc của cây (degree of tree): Bậc của cây là bậc lớn nhất của các nút trên cây. Cây nhị phân là cây có bậc 2, cây nhiều nhánh là cây có bậc lớn hơn 2.

Mức của nút (level of node): Mức của một nút trên cây được định nghĩa như sau:

- Mức của nút gốc là 0.
- Mức của nút khác trong cây nhị phân bằng mức của nút cha + 1.

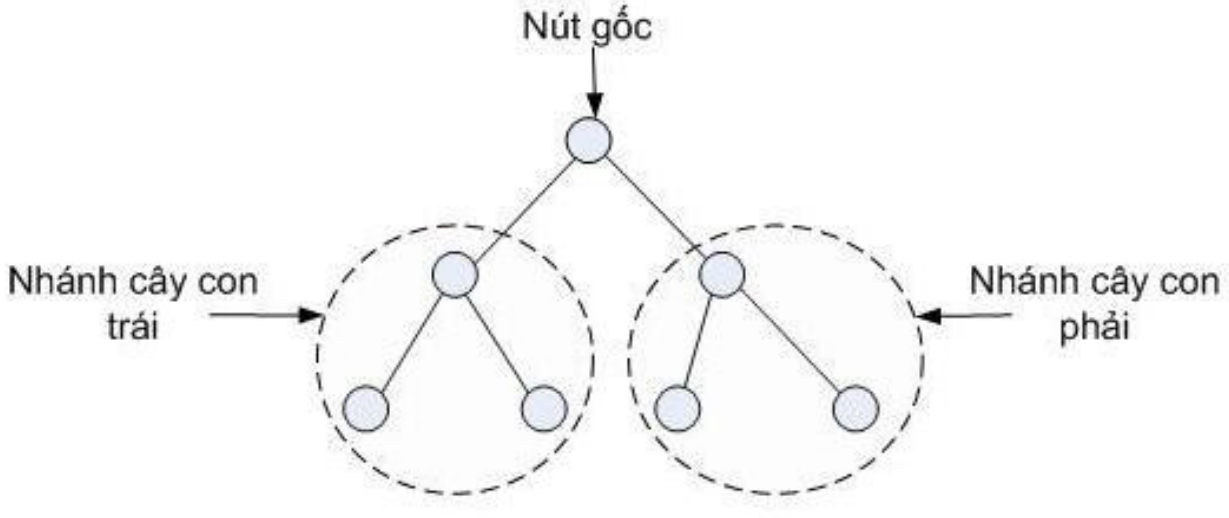
Chiều sâu/độ cao của cây nhị phân (depth of tree): là mức lớn nhất của nút lá trên cây + 1. Chiều sâu chính là đường đi dài nhất từ nút gốc đến nút lá.

Đường đi, chiều dài của đường đi: đường đi là đoạn đường đi từ nút trước đến nút sau. Chiều dài của đường đi = mức của nút sau - mức của nút trước.

4.2 CÂY NHỊ PHÂN

Định nghĩa: Cây nhị phân là một cấu trúc gồm một tập hữu hạn các nút cùng kiểu dữ liệu (tập nút này có thể rỗng) và được phân thành 3 tập con:

- Tập con thứ nhất có một nút gọi là nút gốc (root)
- Hai tập con còn lại tự thân hình thành hai cây nhị phân là nhánh cây con bên trái (left subtree) và nhánh cây con bên phải (right subtree) của nút gốc. Nhánh cây con bên trái hoặc bên phải cũng có thể là cây rỗng.

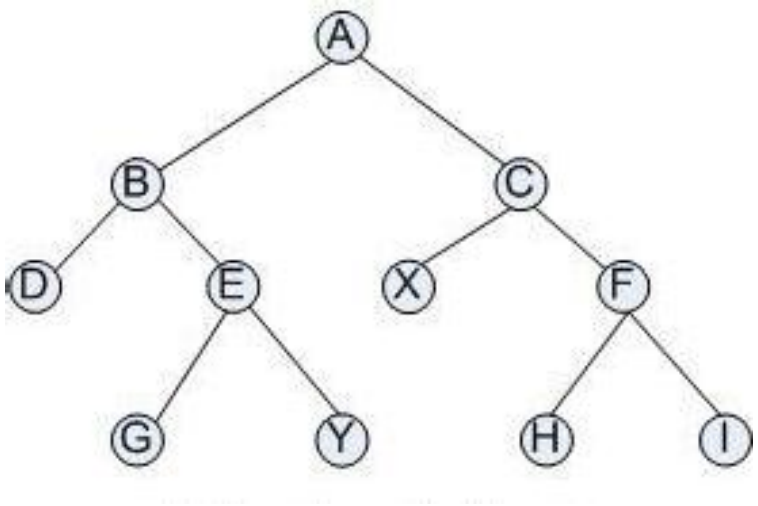


Hình 4.2: Cây nhị phân

Các cây nhị phân đặc biệt

Cây nhị phân đúng (strictly binary tree)

Một cây nhị phân gọi là cây nhị phân đúng nếu nút gốc và tất cả các nút trung gian đều có hai nút con. Nếu cây nhị phân đúng có n nút lá thì cây này sẽ có tất cả 2n - 1 nút.



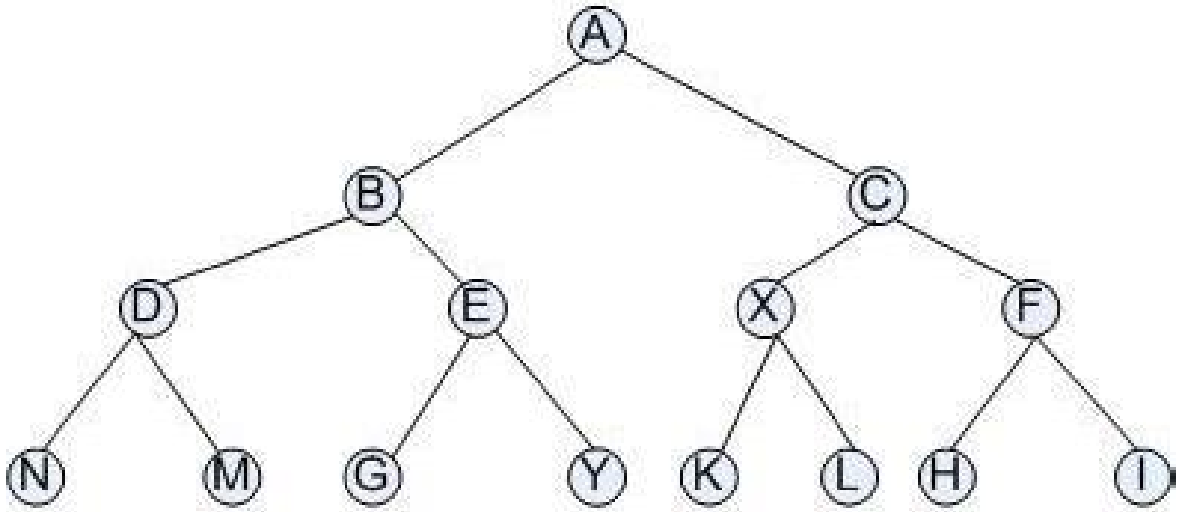
Hình 4.3: Cây nhị phân đúng

Cây nhị phân đầy đủ (complete binary tree)

Một cây nhị phân được gọi là cây nhị phân đầy đủ với chiều sâu d thì:

- Trước tiên nó phải là cây nhị phân đúng.
- Tất cả các nút lá đều có mức là d.

Cây nhị phân đầy đủ là cây nhị phân có số nút tối đa ở mỗi mức.



Hình 4.4: Cây nhị phân đầy đủ.

4.3 MÔ TẢ CÂY NHỊ PHÂN

4.3.1 Mô tả dữ liệu

Cây nhị phân là một cấu trúc gồm một tập hữu hạn các nút cùng kiểu dữ liệu và các nút này được phân thành 3 tập con như sau:

- Tập con thứ nhất chỉ có một nút gọi là nút gốc.
- Hai tập con còn lại tự thân hình thành hai cây nhị phân là nhánh cây con bên trái và nhánh của cây con bên phải của nút gốc. Nhánh cây con bên trái hoặc bên phải có thể rỗng.

4.3.2 Mô tả tác vụ

- Tác vụ *Init*

  Chức năng: khởi động cây nhị phân.

  Dữ liệu nhập: không.

- Tác vụ *IsEmpty*

  Chức năng: Kiểm tra cây có rỗng hay không.

Dữ liệu nhập: Không

Dữ liệu xuất: TRUE|FALSE.

- Tác vụ *CreateNode*

  Chức năng: Cung cấp một nút mới cho cây nhị phân.

  Dữ liệu nhập: nội dung của nút mới x.

  Dữ liệu xuất: Con trỏ chỉ đến nút vừa mới cấp phát.

- Tác vụ *InsertLeft*

  Chức năng: tạo một nút con bên trái (nút lá) của nút p.

  Dữ liệu nhập: Con trỏ chỉ nút p và nội dung của nút x.

  Điều kiện: nút p chưa có nút con bên trái.

  Dữ liệu xuất: không.

- Tác vụ *InsertRight*

  Chức năng: tạo nút con bên phải (nút lá) của nút p.

  Dữ liệu nhập: Con trỏ chỉ nút p và nội dung của nút x.

  Điều kiện: Nút p chưa có nút con bên phải.

  Dữ liệu xuất: không.

- Tác vụ *DeleteLeft*

  Chức năng: xoá nút con bên trái (nút lá) của nút p.

  Dữ liệu nhập: con trỏ chỉ nút p.

  Điều kiện: nút con trái của nút p là nút lá.

  Dữ liệu xuất: nút bị xoá.

- Tác vụ *DeleteRight*

  Chức năng: xoá nút con bên phải (nút lá) của nút p.

  Dữ liệu nhập: con trỏ chỉ nút p.

  Điều kiện: nút con phải của nút p là nút lá.

  Dữ liệu xuất: nút bị xoá.

- Tác vụ *PreOrder*

  Chức năng: duyệt cây theo thứ tự trước (NLR).

  Dữ liệu nhập: không.

  Dữ liệu xuất: Không.

- Tác vụ *InOrder*

  Chức năng: duyệt cây theo thứ tự giữa (LNR)

  Dữ liệu nhập: Không.

  Dữ liệu xuất: Không.

- Tác vụ *PostOder*

  Chức năng: duyệt cây theo thứ tự sau (LRN)

  Dữ liệu nhập: Không.

  Dữ liệu xuất: Không.

- Tác vụ *Search*

  Chức năng: tìm kiếm nút trong cây nhị phân theo một khoá tìm kiếm.

  Dữ liệu nhập: khoá tìm kiếm.

  Dữ liệu xuất: con trỏ chỉ nút tìm thấy.

- Tác vụ *ClearTree*

  Chức năng: dùng để xoá cây nhị phân.

  Dữ liệu nhập: Không.

  Dữ liệu xuất: Không.

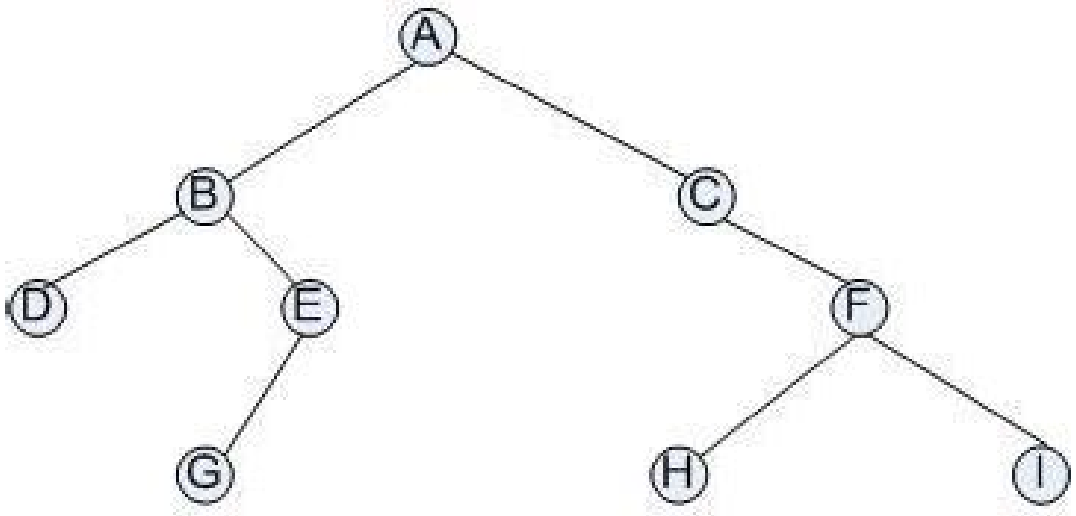
4.3.3 Duyệt cây nhị phân

Có hai phép duyệt cây nhị phân chính đó là duyệt theo chiều rộng và duyệt theo chiều sâu. Duyệt cây theo chiều rộng là cách duyệt cây theo một mức hay độ sâu nhất định trước khi duyệt tới mức tiếp theo sâu hơn. Các bước thực hiện khi duyệt cây theo chiều rộng như sau: Đầu tiên duyệt nút gốc (mức 0), sau đó duyệt các nút ở mức 1, 2, ...

Đối với phép duyệt cây theo chiều sâu, có ba cách như sau:

- *PreOder*: duyệt cây theo thứ tự trước (NLR - Node Left Right hoặc NRL – Node Right Left). Đầu tiên thăm nút gốc, sau đó đến duyệt cây con bên trái (phải), sau đó duyệt cây con bên phải (trái).
- *InOder*: duyệt cây theo thứ tự giữa (LNR – Left Node Right hoặc RNL – Right Node Left): Đầu tiên duyệt qua nhánh cây con bên trái (phải), sau đó thăm nút gốc, cuối cùng duyệt cây con bên phải (trái).
- *PostOder*: Duyệt cây theo thứ tự sau (LRN – Left Right Node hoặc RLN – Right Left Node): Đầu tiên, duyệt nhánh cây con bên trái (phải), sau đó duyệt nhánh cây con bên phải (trái), cuối cùng thăm nút gốc.

Hình vẽ sau đây mô tả ví dụ của ba phép duyệt cây nhị phân:



Hình 4.5: Minh họa duyệt cây

Nếu duyệt cây trên theo chiều rộng thì thứ tự các nút sẽ là: A B C D E F G H I

Nếu duyệt cây trên theo thứ tự NLR thì thứ tự các nút sẽ là: A B D E G C F H I.

Nếu duyệt cây trên theo thứ tự LNR thì thứ tự các nút là: D B G E A C H F I.

Nếu duyệt cây trên theo thứ tự LRN thì thứ tự các nút là: D G E B H I F C A.

4.4 HIỆN THỰC CÂY NHỊ PHÂN TỔNG QUÁT

4.4.1 Khai báo cấu trúc của một nút

Mỗi nút trên cây nhị phân tổng quát là một mẩu tin có các trường như sau:

- Trường info: chứa nội dung của nút.
- Trường left là con trỏ chỉ nút, dùng để chỉ nút con bên trái.
- Trường right là con trỏ chỉ nút, dùng để chỉ nút con bên phải.

```
typedef int DataType;
typedef struct nodetype{
    DataType info;
    struct nodetype *left;
   struct nodetype *right;
}Node;
typedef Node *NODEPTR;
```

4.4.2 Hiện thực các tác vụ

(minh họa với *info* kiểu số nguyên)

- Tác vụ *CreateNode*: Tác vụ này dùng để cấp phát một nút mới có dữ liệu là x.

```
    NODEPTR CreateNode(DataType x){
            NODEPTR p=new Node;
            p->info=x;
            p->left=NULL;
            p->right=NULL;
    return p;
    }
```

- Tác vụ *PreOrder*

```
void PreOder(NODEPTR proot){
    if(proot !=NULL){
            printf("%4d",proot->info);
            PreOder (proot->left)
            PreOder (proot->right);
    }
}
```

- Tác vụ *InOder*

```
void InOder (NODEPTR proot){
    if(proot!=NULL){
            InOder (proot->left);
            printf("%4d",proot->info);
            InOder (proot->right);
    }
}
```

- Tác vụ *PostOder*

```
void PostOder (NODEPTR proot){
    if(proot!=NULL){
            PostOder (proot->left);
            PostOder (proot->right);
            printf("%4d", proot->info);
    }
}
```

Lưu ý: Trong các hàm *PreOrder, InOrder, PostOrder*, thao tác xuất (dòng lệnh printf("%4d",proot->info)) chỉ mang tính minh họa. Tùy vào mục đích duyệt cây, thao tác xuất có thể được thay thế bởi thao tác khác.

- Tác vụ *Search*

```
NODEPTR Search(NODEPTR proot, DataType x){
    NODEPTR p;
    if(proot->info==x)   return proot;
    if(proot==NULL)      return NULL;
    p=Search(proot->left, x);
    if(p==NULL)
            p=Search(proot->right, x);
   return p;
}
```

- Tác vụ *ClearTree*

```
void ClearTree(NODEPTR &proot){
    if(proot!=NULL){
            ClearTree(proot->left);
            ClearTree(proot->right);
            delete proot;
    }
}
```

- Tác vụ *InsertLeft*

```
void InsertLeft (NODEPTR p, DataType x){
        if(p==NULL)
                printf("\n Nut khong ton tai");
        else
                if(p->left !=NULL)
                        printf("\n Nut p da co con ben trai");
                else
                        p->left= CreateNode(x);
}
```

- Tác vụ InsertRight

```
void InsertRight(NODEPTR p, DataType x){
    if(p==NULL)
            printf("\n Nut khong ton tai");
    else
            if(p->right!=NULL)
                    printf("\n Nut da co con ben phai");
            else
            p->right=CreateNode(x);
}
```

- Tác vụ DeleteLeft

```
int DeleteLeft(NODEPTR p){
    NODEPTR q;
    int x;
    if(p==NULL){
    printf("\n Nut khong ton tai");
    }
    else{
            q=p->left;
            x=q->info;
            if(q==NULL)
                    printf("\n Nut khong co con ben trai");
            else{
                    if(q->left!=NULL ||q->right !=NULL)
                            printf("\n Nut khong phai la la");
                    else{
                            p->left=NULL;
                            delete q;
                    }
            }
    }
   return x;
}
```

- Tác vụ *DeleteRight*

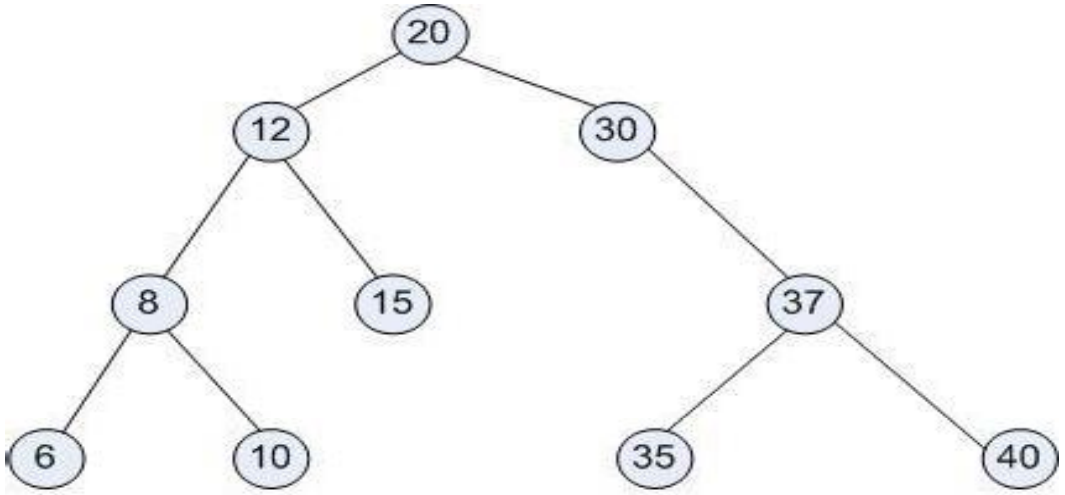
```
int DeleteRight(NODEPTR p){
    NODEPTR q;
    int x;
    if(p==NULL){
    printf("\n Nut khong ton tai");
    }
    else{
            q=p->right;
            x=q->info;
            if(q==NULL)
                    printf("\n Nut khong co con ben phai");
            else{
                    if(q->left!=NULL ||q->right !=NULL)
                            printf("\n Nut khong phai la la");
                    else{
                            p->right=NULL;
                            delete q;
                    }
            }
    }
   return x;
}
```

4.5 ĐỊNH NGHĨA CÂY NHỊ PHÂN TÌM KIẾM

Cây nhị phân tìm kiếm (*Binary Search Tree*, *BST*) là cây nhị phân hoặc bị rỗng, hoặc tất cả các nút trên cây có nội dung thoả mãn các điều kiện sau:

- Nội dung của tất cả các nút thuộc nhánh cây con bên trái đều nhỏ hơn nội dung của nút gốc.
- Nội dung của tất cả các nút thuộc nhánh cây con bên phải đều lớn hơn nội dung của nút gốc.
- Cây con bên trái và cây con bên phải tự thân cũng hình thành hai cây nhị phân tìm kiếm.

Hình vẽ sau đây mô tả cây nhị phân tìm kiếm.



Hình 4.6: Cây nhị phân tìm kiếm.

Ưu điểm của cây nhị phân tìm kiếm

Trong phần này, ta sẽ so sánh các đặt điểm của cây nhị phân tìm kiếm với danh sách liên kết và danh sách kề dựa trên 2 tiêu chí là việc tìm kiếm dữ liệu và việc cập nhật dữ liệu.

- *Với danh sách kề:*

Tác vụ thêm nút, xoá nút trên danh sách kề không hiệu quả vì chúng phải dời chỗ nhiều lần các nút trong danh sách. Tuy nhiên, nếu danh sách kề là có thứ tự thì tác vụ tìm kiếm trên danh sách thực hiện rất nhanh bằng phương pháp tìm kiếm nhị phân, tốc độ tìm kiếm tỉ lệ với O(log n).

- *Với danh sách liên kết:*

Tác vụ thêm nút, xoá nút trên danh sách liên kết rất hiệu quả, lúc này chúng ta không phải dời chỗ các nút mà chỉ hiệu chỉnh một vài liên kết cho phù hợp. Nhưng tác vụ tìm kiếm trên danh sách liên kết không hiệu quả vì thường dùng phương pháp tìm kiếm tuyến tính dò từ đầu danh sách. Tốc độ tìm kiếm tỉ lệ với O(n).

- *Cây nhị phân tìm kiếm:*

Là cấu trúc dung hoà được 2 yếu tố trên: việc thêm nút hay xoá nút trên cây khá thuận lợi và thời gian tìm kiếm khá nhanh. Nếu cây nhị phân tìm kiếm là cân bằng thì thời gian tìm kiếm là O(log(n)), với n là số phần tử trên cây.

4.6 CÀI ĐẶT CÂY NHỊ PHÂN TÌM KIẾM

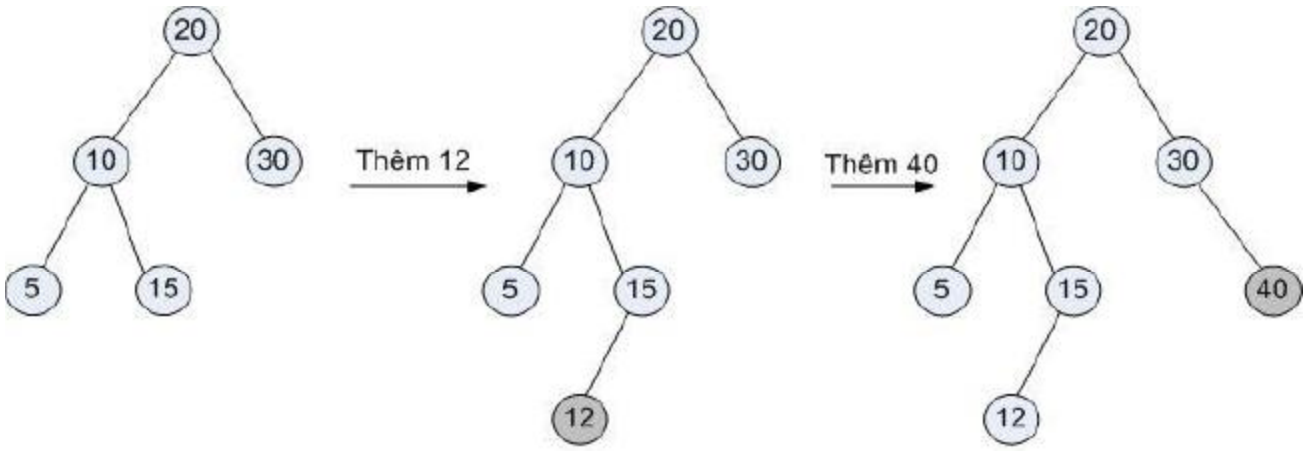
Cây nhị phân tìm kiếm là một dạng đặc biệt của cây nhị phân nên chúng ta vẫn dùng các tác vụ trong phần trên hiện thực cho cây nhị phân tìm kiếm. Ở phần này chúng ta chỉ xét các tác vụ tìm kiếm, thêm vào cây một phần tử và xoá một phần tử ra khỏi cây nhị phân tìm kiếm.

Tác vụ tìm kiếm (Search) trên cây BST:

```
NODEPTR search(NODEPTR proot, DataType x){
    NODEPTR p;
    p=proot;
    if(p!=NULL)
            if(x<proot->info)      p=search(proot->left, x);
            else
                    if(x>proot->info)      p=search(proot->right, x);
    return p;
}
```

Tác vụ thêm một phần tử vào cây BST

Hình vẽ sau đây mô tả việc thêm 2 nút có nội dung là 12 và 40 vào cây nhị phân tìm kiếm.



Hình 4.7: Tác vụ thêm một phần tử vào cây BST.

Sau đây là hiện thực tác vụ thêm một phần tử vào cây nhị phân tìm kiếm dùng phương pháp đệ qui.

```
void Insert(NODEPTR &proot, DataType x) {
    if (isEmpty(proot)  //nếu cây rỗng, thêm trực tiếp vào cây
            proot = CreateNode(x);
    else
            if (x==proot->info) //nếu đã có x trong cây
                    return;
            if (x<proot->info)
                    Insert(proot->left, x);
            else
                    Insert(proot->right, x);
}
```

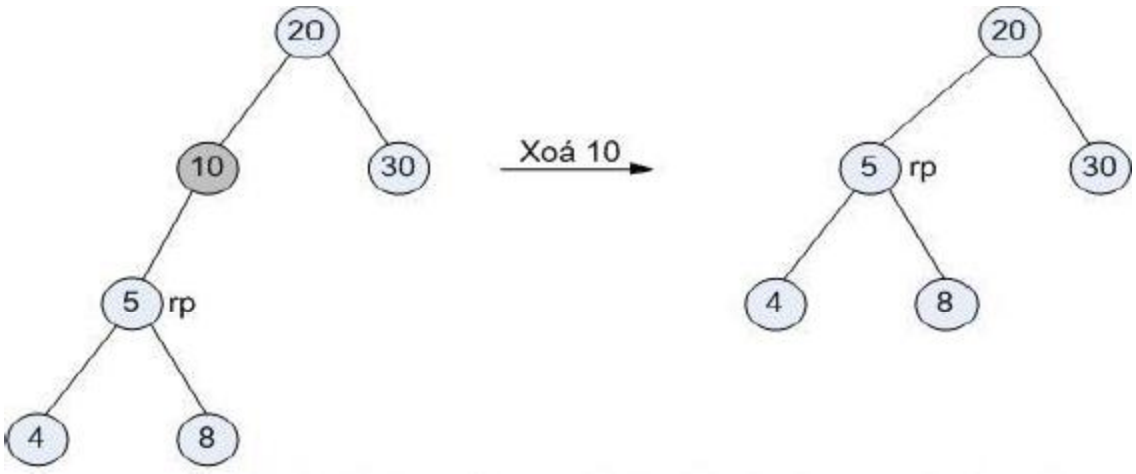
Tác vụ xoá một phần tử ra khỏi cây BST

Việc xoá một nút p trong cây nhị phân tìm kiếm khá phức tạp, vì khi đó chúng ta phải điều chỉnh lại cây sao cho nó vẫn là cây nhị phân tìm kiếm. Có 3 trường hợp cần chú ý khi xoá một phần tử trên cây nhị phân tìm kiếm.

- *Trường hợp 1:* Nếu nút p cần xoá là nút lá, việc xoá nút lá chỉ đơn giản là việc huỷ nút lá đó.

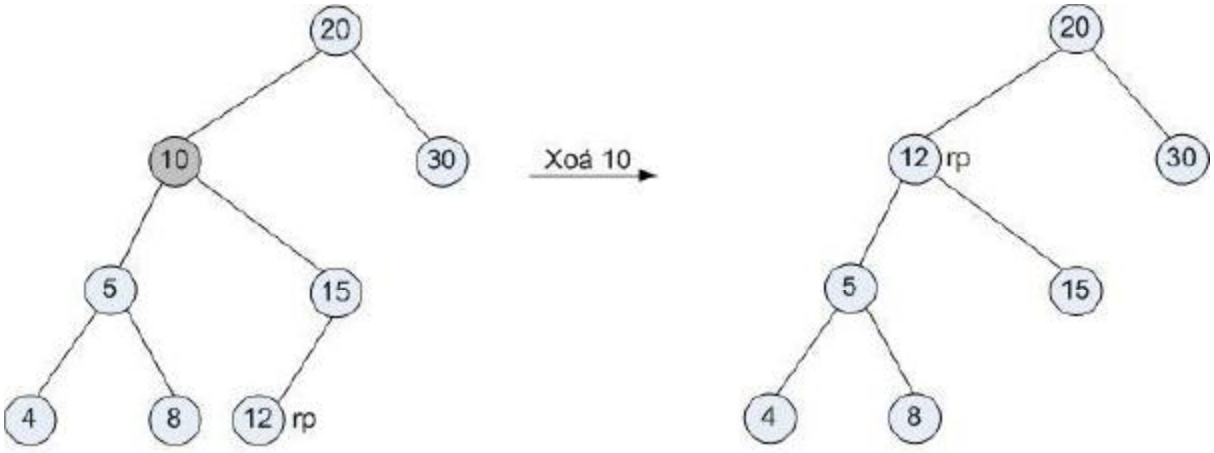
Hình 4.8: Xóa nút lá trên cây BST.

- *Trường hợp 2:* Nếu nút p cần xoá có một cây con, chúng ta chọn nút con của p làm nút thế mạng cho nút p. Chúng ta phải tạo liên kết từ nút cha của p đến nút thế mạng, sau đó huỷ nút p đi.



Hình 4.9: Xóa nút có một cây con trên cây BST.

- *Trường hợp 3:* Nếu nút p cần xoá có hai cây con, chúng ta chọn nút có nội dung gần p nhất làm nút thế mạng cho nút p. Nút thế mạng cho nút p có thể là nút trái nhất của nhánh cây con bên phải nút p hoặc là nút phải nhất của cây con bên trái nút p.



Hình 4.10: Xóa nút có hai cây con trên cây BST.

```
int Remove(NODEPTR &proot, DataType x) {
    if ( proot == NULL)
            return FALSE; //không tìm thấy nút cần xoá
    if (proot->info >x)    //tìm và xóa bên trái
            return Remove(proot->left, x);
    if (proot->info <x)    //tìm và xóa bên phải
            return Remove(proot->right, x);
//nếu (proot->info==x)
    Node* p, f, rp;
```

```
p = proot;              //p biến tạm trỏ đến proot
//trường hợp proot có 1 cây con
if ( proot->left == NULL)    //có 1 cây con
        proot = proot->right;
else if (proot->right == NULL)      //có 1 cây con
        proot = proot->left;
else { //TH pTree có 2 cây con chọn nút nhỏ nhất bên cây con phải
        f = p;  //f để lưu cha của rp
        rp = p->right;          //rp bắt đầu từ p->right
        while ( rp->left != NULL)    {
                f = rp;  //lưu cha của rp
                rp = rp->left; //rp qua bên trái
        } //kết thúc khi rp là nút có nút con trái là null
        p->info = rp->info; //đổi giá trị của p và rp
        if ( f == p) //nếu cha của rp là p
                f->right = rp->right;
        else //f != p
                f->left = rp->right;
        p = rp; // ptrỏ đến phần tử thế mạng rp
}
delete p;               //xoá nút p
return TRUE;
```

TÓM TẮT

Trong bài này, học viên cần nắm:

Cây nhị phân là một cấu trúc gồm một tập hữu hạn các nút cùng kiểu dữ liệu tổ chức theo cấu trúc phân cấp.

Các thao tác cơ bản trên cây: Init, IsEmpty, InsertLeft, InsertRight, DeleteLeft, DeleteRight, PreOder, InOder, PostOder, Search, ClearTree.

Cây nhị phân tìm kiếm BST là cây nhị phân mà mỗi nút thoả: Giá trị của tất cả nút con trái nhỏ hơn giá trị của nút đó, và giá trị của tất cả nút con phải lớn hơn giá trị của nút đó.

Cây nhị phân tìm kiếm là một dạng đặc biệt của cây nhị phân nên vẫn dùng các tác vụ trong phần trên hiện thực cho cây nhị phân tìm kiếm, chỉ khác ở các tác vụ: tìm kiếm, thêm vào cây một phần tử và xoá một phần tử ra khỏi cây nhị phân tìm kiếm

CÂU HỎI ÔN TẬP

Câu 1: Cho trước 1 mảng a có n phần tử (mảng số nguyên/ mảng cấu trúc có một trường là khóa), hãy tạo một cây nhị phân có n node, mỗi nút lưu 1 phần tử của mảng.

a. Cài đặt hàm duyệt cây theo thứ tự: LNR, NLR, LRN, mức.

b. Tìm node có giá trị là X.

c. Xác định chiều cao của cây

d. Đếm số node trên cây.

e. Đếm số node lá

f. Đếm số node thỏa ĐK: đủ 2 cây con, có giá trị nhỏ hơn K, có giá trị lớn hơn giá trị của node con trái và nhỏ hơn giá trị của node con phải, có chiều cao cây con trái bằng chiều cao cây con phải.

g. Đếm số node lá có giá trị > X.

h. Cho biết node có giá trị lớn nhất/ nhỏ nhất.

i. Kiểm tra cây có cân bằng không?

j. Kiểm tra có là cây cân bằng hoàn toàn hay không?

k. Kiểm tra có phải là cây "đẹp" hay không? (mọi nút đều có đủ 2 nút con trừ nút lá).

l. Cho biết mọi node trên cây đều nhỏ hơn X (hoặc lớn hơn X) hay không?

Câu 2: Xét tác vụ insert và remove để thêm nút và xoá nút trên cây nhị phân tìm kiếm.

a. Vẽ lại hình ảnh của cây BST nếu thêm các nút vào cây theo thứ tự như sau: 8, 3, 5, 2, 20, 11, 30, 9, 18, 4.

b. Vẽ lại hình ảnh của cây trên nếu ta lần lược xoá 2 nút 5 và 20.

Câu 3: Cài đặt cấu trúc dữ liệu liên kết cho cây nhị phân tìm kiếm, với các thao tác:

a. Cài đặt các thao tác xây dựng cây: Init, IsEmpty, CreateNode

b. Cài đặt thao tác cập nhật: Insert, Remove, ClearTree

c. Xuất danh sách tăng dần và giảm dần

d. Kiểm tra xem cây có phải là cây nhị phân đúng

e. Kiểm tra xem cây có phải là cây nhị phân đầy đủ

f. Xác định nút cha của nút chứa khoá x

g. Đếm số nút lá, nút giữa, kích thước của cây

h. Xác định độ sâu/chiều cao của cây

i. Tìm giá trị nhỏ nhất/lớn nhất trên cây

j. Tính tổng các giá trị trên cây.

BÀI 5: CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG - AVL

Sau khi học xong bài này, học viên có thể:

- *Hiểu được cấu trúc cây nhị phân tìm kiếm cân bằng.*
- *Nắm được các trường hợp khi thêm xóa nút làm cây mất cân bằng, cân bằng lại cây.*
- *Cài đặt được các thao tác trên cây AVL.*
- *Vận dụng cấu trúc cây AVL để giải các bài toán cụ thể.*

5.1 ĐỊNH NGHĨA CÂY NHỊ PHÂN TÌM KIẾM CÂN BẰNG

Người ta dùng cây nhị phân tìm kiếm với mục đích thực hiện tác vụ tìm kiếm cho nhanh, tuy nhiên để tìm kiếm trên cây nhanh thì cây cần phải cân đối. Trường hợp tối ưu nhất là cây nhị phân tìm kiếm hoàn toàn cân bằng (là cây có sự khác biệt của tổng số nút cây con bên trái và tổng số nút của cây con bên phải không quá 1). Tuy nhiên khi thêm vào hoặc xoá nút trên cây rất dễ làm cây mất cân bằng, và chi phí để cân bằng lại cây rất lớn vì phải thao tác trên toàn bộ cây.

Do vậy, người ta tìm cách tổ chức một cây nhị phân tìm kiếm đạt trạng thái cân bằng yếu hơn nhằm làm giảm thiểu chi phí cân bằng khi thêm nút hay xoá nút. Một dạng cây cân bằng là cây nhị phân tìm kiếm cân bằng do hai nhà toán học người Nga là **A**delson **V**elski và **L**andis xây dựng vào năm 1962 nên còn được gọi là cây AVL. Cây AVL là cây nhị phân tìm kiếm mà tại tất cả các nút của nó chiều sâu của cây con bên phải và chiều sâu của cây con bên trái chênh nhau không quá 1.

Gọi lh(p) và rh(p) là chiều sâu của cây con bên trái và chiều sâu của cây con bên phải của nút p. Có 3 trường hợp có thể xảy ra đối với cây AVL:

- lh(p)=rh(p): nút p cân bằng.

- lh(p)=rh(p) + 1: nút p bị lệch về bên trái.
- lh(p)=rh(p) – 1: nút p bị lệch về bên phải.

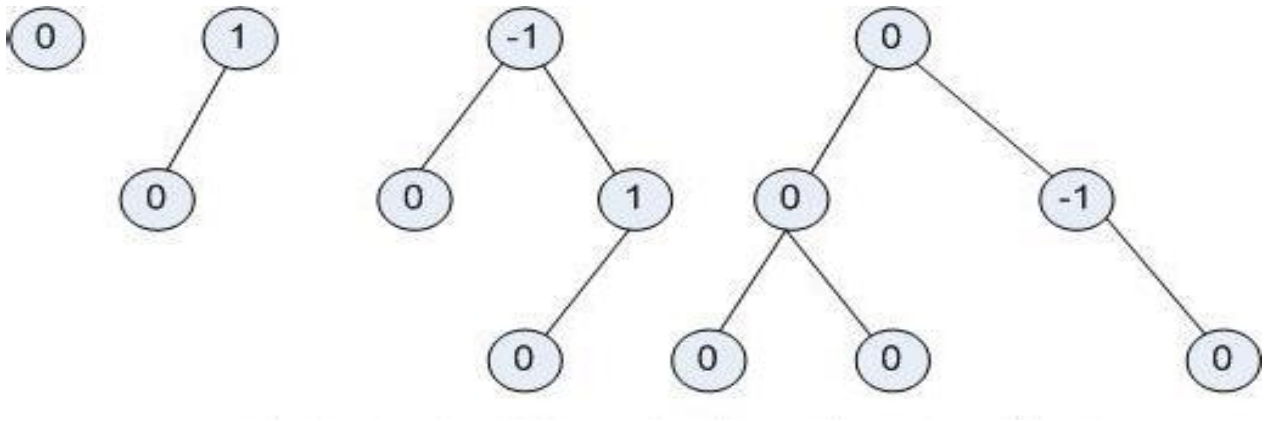
Với cây AVL, khi thêm nút hay xoá nút trên cây có thể làm tăng hay giảm chiều sâu của cây nên có thể làm cây AVL mất cân bằng, khi đó chúng ta phải cân bằng lại cây. Tuy nhiên việc cân bằng lại trên cây AVL chỉ xảy ra ở phạm vi cục bộ bằng cách xoay trái hay xoay phải ở một vài nhánh cây con nên giảm thiểu chi phí cân bằng.

Chỉ số cân bằng (balance factor) bf của một nút trên cây AVL là hiệu của chiều sâu cây con bên trái và chiều sâu cây con bên phải của nút đó.

Xét một nút p bất kỳ trên cây AVL, chỉ số cân bằng của nút p chỉ có thể là một trong 3 giá trị sau:

- bf(p)=0: lh(p)=rh(p) nút p cân bằng
- bf(p)=1: lh(p)=rh(p) + 1 nút p bị lệch trái
- bf(p)=-1: lh(p)=rh(p) – 1 nút p bị lệch phải.

Hình vẽ sau đây minh hoạ các cây AVL, mỗi nút có một số là chỉ số cân bằng của nút đó.



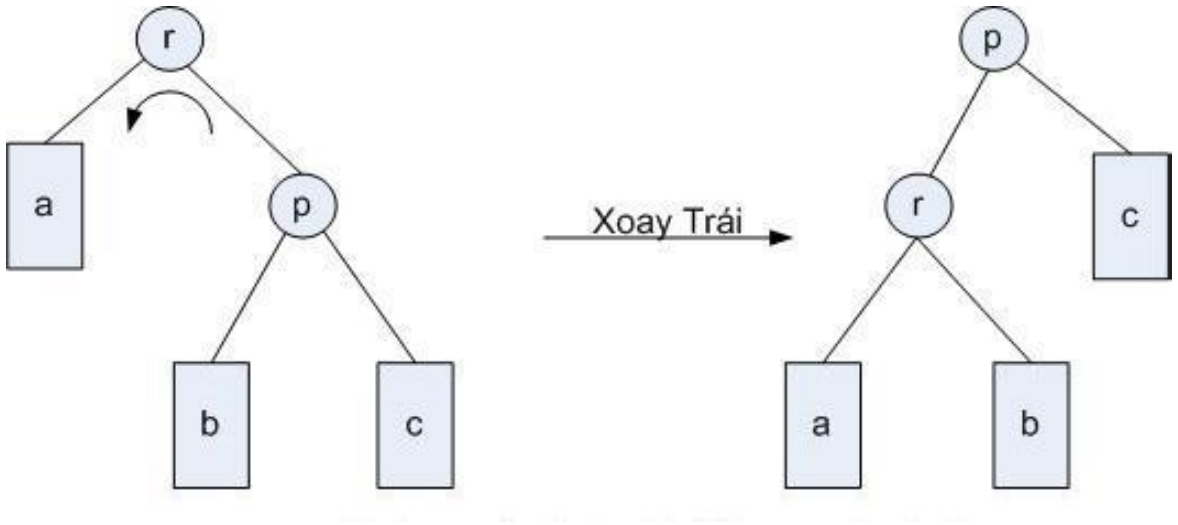
Hình 5.1: Các cây AVL và chỉ số cân bằng của mỗi nút

5.2 CÁC TÁC VỤ XOAY

Khi thêm vào hoặc xoá đi một nút trên cây AVL, cây có thể mất cân bằng. Để cân bằng lại cây, chúng ta sẽ dùng các tác vụ xoay trái và xoay phải. Trong phần này chúng ta sẽ nghiên cứu hai phép xoay trái và xoay phải.

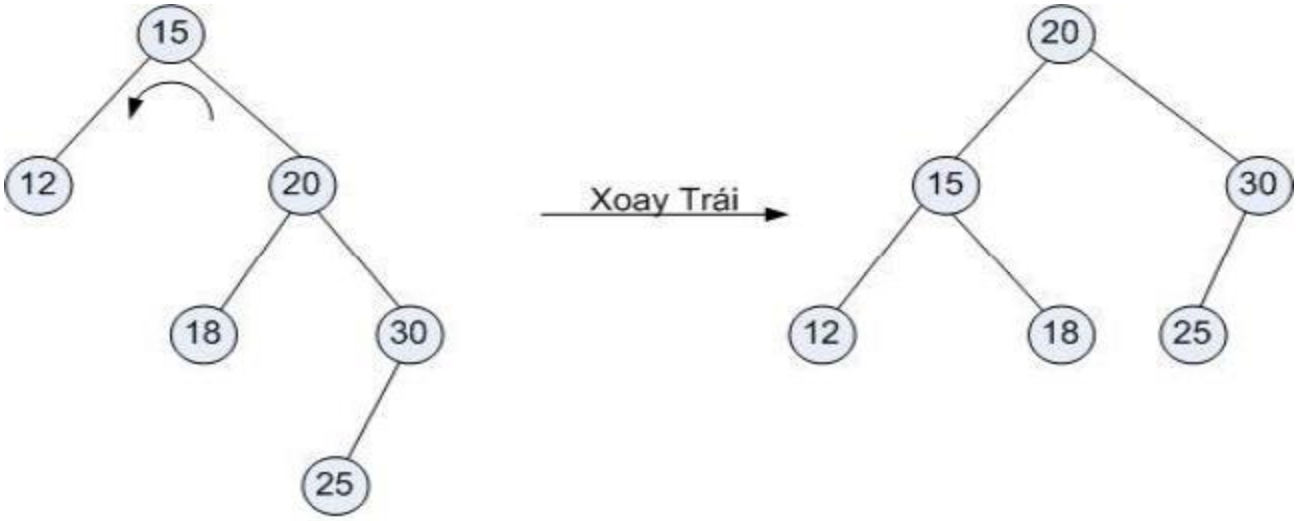
5.2.1 Tác vụ xoay trái (RotateLeft)

Hình vẽ sau mô tả tác vụ xoay trái cây nhị phân tìm kiếm quanh nút r, yêu cầu là nút r phải có nút con bên phải gọi là nút p. Sau khi xoay xong, nút p thành gốc của cây nhị phân tìm kiếm.



Hình 5.2: Xoay trái cây nhị phân quanh gốc là r

Hình vẽ sau minh hoạ cho việc xoay trái quanh cây nhị phân tìm kiếm có nút gốc là 15 như sau:



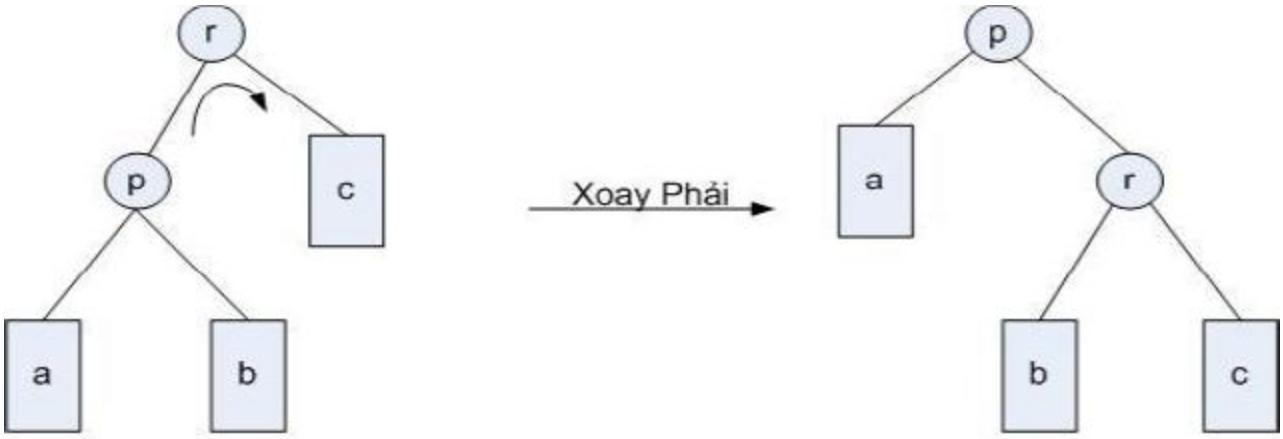
Hình 5.3: Xoay trái cây nhị phân quanh gốc là 15.

Đoạn code sau sẽ minh hoạ tác vụ xoay trái quanh nút gốc proot, tác vụ này trả về con trỏ chỉ nút gốc mới của nhánh.

```
NODEPTR RotateLeft(NODEPTR proot){
    NODEPTR p;
    p=proot;
    if(proot==NULL){
        printf("\n Khong the xoay trai vi cay rong");
    }else{
        if(proot->right==NULL)
            printf("\n Khong the xoay trai vi khong co node con ben phai");
        else{
            p=proot->right;
            proot->right=p->left;
            p->left=proot;
        }
    }
  return p;
}
```

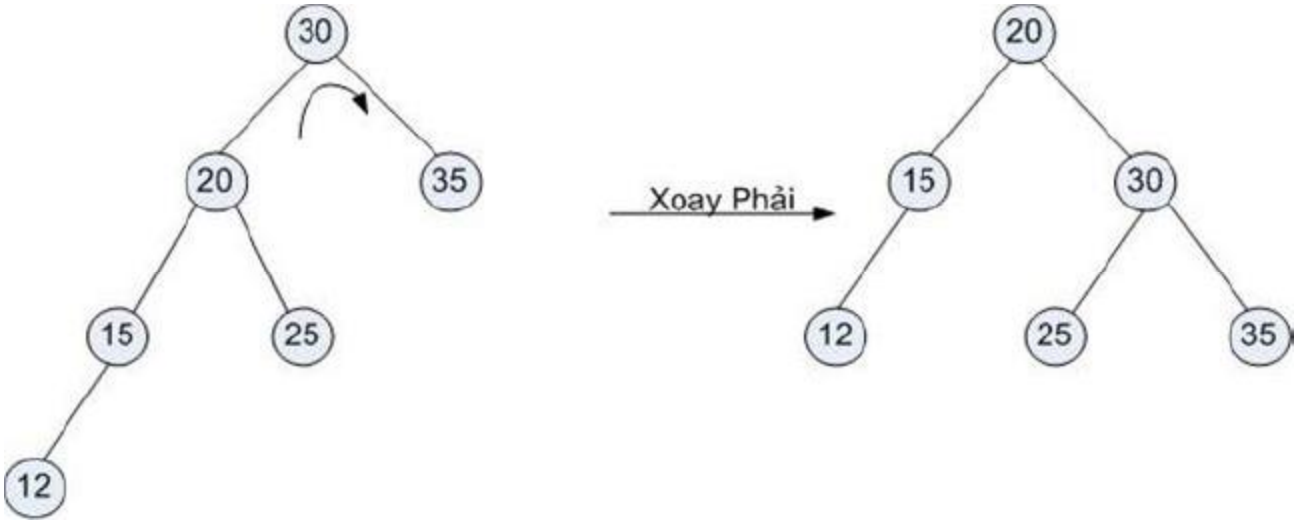
5.2.2 Tác vụ xoay phải (RotateRight)

Hình vẽ sau mô tả tác vụ xoay phải quanh nút gốc r.



Hình 5.4: Xoay phải cây nhị phân quanh gốc là r.

Hình vẽ sau minh hoạ cho việc xoay phải quanh nút gốc của cây nhị phân tìm kiếm:



Hình 5.5: Xoay phải cây nhị phân quanh gốc là 30.

Đoạn mã sau đây hiện thực tác vụ xoay phải quanh nút proot, tác vụ này trả về con trỏ chỉ nút gốc mới của nhánh.

```
NODEPTR RotateRight(NODEPTR proot){
    NODEPTR p;
    p=proot;
    if(proot==NULL)
            printf("\n Khong the xoay phai vi cay bi rong");
    else
            if(proot->left==NULL)
                    printf("\n Khong the xoay phai vi khong co con ben trai");
            else{
                    p=proot->left;
                    proot->left=p->right;
          p->right=proot;
              }
    return p;
}
```

5.3 THÊM MỘT NÚT VÀO CÂY AVL

Việc thêm một nút vào cây AVL khá phức tạp, được tiến hành qua các bước sau:

- Trước tiên chúng ta thêm nút vào cây AVL như thêm nút vào cây nhị phân tìm kiếm, nghĩa là nút mới thêm vào sẽ là nút lá ở vị trí thích hợp trên cây.

- Tiếp theo chúng ta tính lại chỉ số cân bằn của các nút có bị ảnh hưởng.
- Sau đó chúng ta xét cây có bị mất cân bằng không, nếu cây bị mất cân bằng thì phải cân bằng lại cây (bằng cách thực hiện các phép xoay phù hợp)

Trường hợp thêm vào làm cây mất cân bằng:

Có hai trường hợp khi thêm nút vào thì cây sẽ bị mất cân bằng.

- Cây sẽ bị mất cân bằng nếu vị trí thêm nút là vị trí sau bên trái của một nút trước gần nhất bị lệch trái.
- Cây sẽ bị mất cân bằng nếu vị trí thêm nút là vị trí sau bên phải của một nút trước gần nhất bị lệch phải.

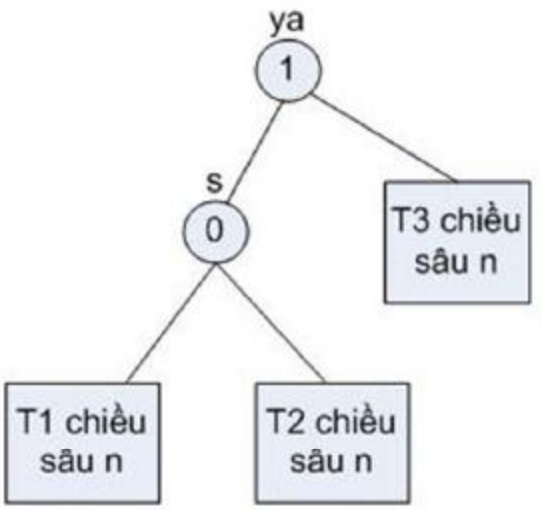
Thực hiện các phép xoay để cân bằng lại cây

Nếu khi thêm nút làm cây bị mất cân bằng, người ta sẽ thực hiện các phép xoay phù hợp để đưa cây trở về trạng thái cân bằng. Gọi ya là nút trước gần nhất bị mất cân bằng khi thêm nút x vào cây AVL, vấn đề là phải xoay cây như thế nào trong 2 trường hợp sau:

Trường hợp 1: bf(ya)=1, nút lá thêm vào cây là nút sau bên trái ya.

Trường hợp 2: bf(ya)=-1, nút lá thêm vào cây là nút sau bên phải ya.

Vì hai trường hợp là tương tự nhau nên ở đây ta chỉ xét trường hợp 1, còn trường hợp 2 được suy ra dựa trên trường hợp 1.

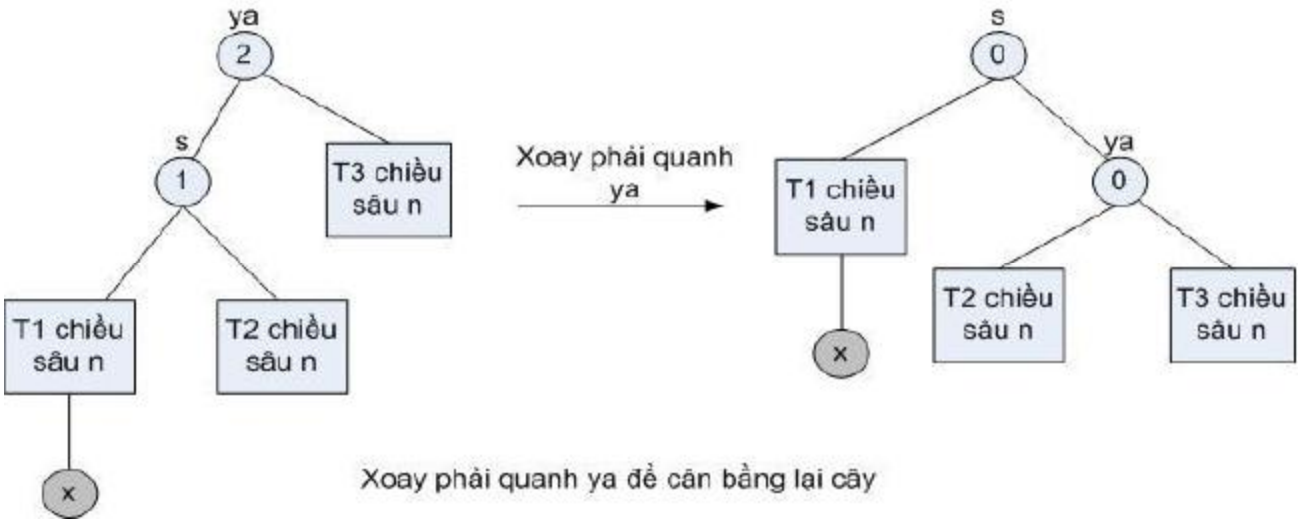


Hình 5.6: Cây con nút gốc ya trước khi thêm

Trong trường hợp 1, xét nhánh cây có nút gốc ya:

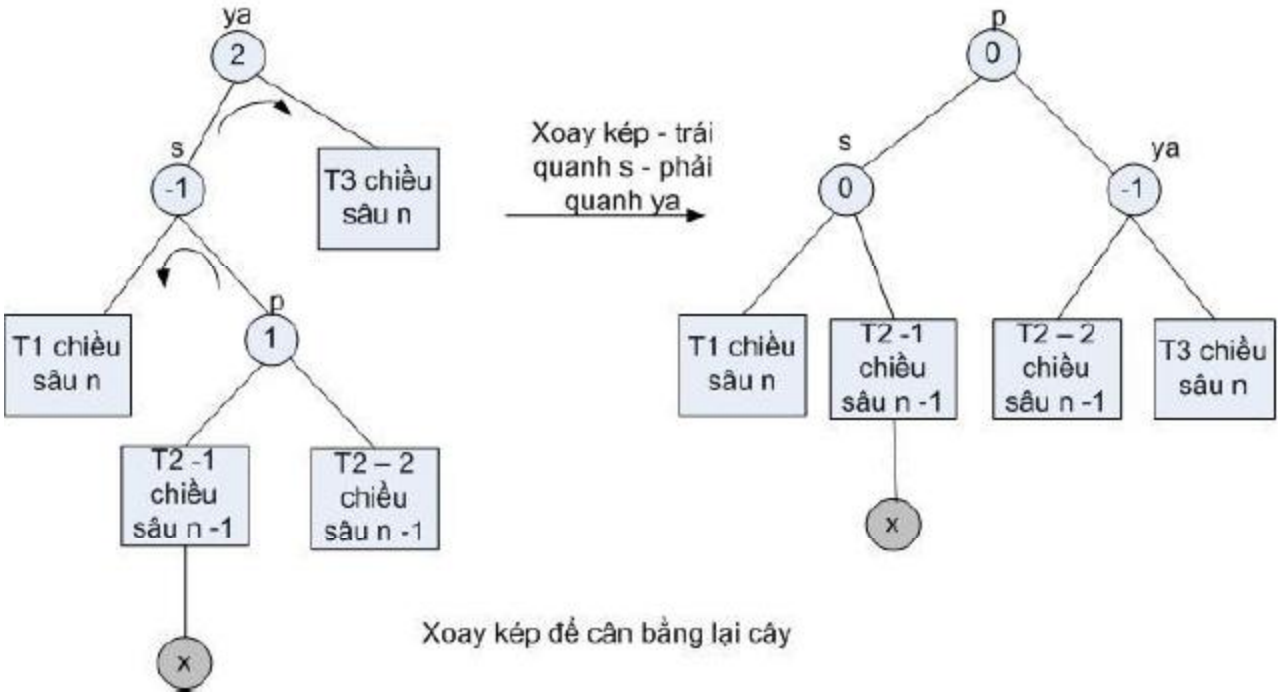
Khi thêm nút x vào nhánh cây con trên, có 2 vị trí thêm phải cân bằng lại là thêm vào nhánh T1 và thêm vào ở nhánh T2.

Thêm vào ở nhánh T1:



Hình 5.7: Xoay phải quanh ya.

Thêm vào ở nhánh T2: Phải tiến hành xoay kép.



Hình 5.8: Xoay kép.

Trường hợp cây bị lệch phải và thêm một nút vào nhánh cây con bên phải thì làm tương tự như trường hợp trên.

5.4 CÀI ĐẶT CÂY AVL

5.4.1 Khai báo cấu trúc cho cây AVL

Khi khai báo nút của cây AVL, ta cần khai báo thêm một trường bf (balance factor) cho biết chỉ số cân bằng của nút.

```
struct nodetype{
    DataType info;
    int bf; //balance factor
  struct nodetype *left, *right;
};
typedef struct nodetype *NODEPTR;
```

5.4.2 Các tác vụ của cây AVL

Phần lớn các tác vụ được dùng lại từ những phần trên trừ tác vụ thêm một nút vào cây AVL. Nên phần này ta chỉ xét tác vụ thêm một phần tử vào cây AVL.

```
void Insert(NODEPTR *pavltree, DataType x)
{
    //fp la nut cha cua p, q la con cua p
    //ya la nut truoc gan nhat co the mat can bang, fya la cha cua ya
    //s la nut con cua ya theo huong mat can bang
    //imbal=1: lech trai; =-1 lech phai
    NODEPTR fp, p, q, fya, ya, s;
    int imbal;
   //khoi dong cac gia tri
    fp=NULL;
    p=*pavltree;
    fya=NULL;
    ya=p;
    while(p!=NULL){
            if(x==p->info)      return;
            if(x<p->info) q=p->left;
            if(x>p->info) q=p->right;
            if(q!=NULL)
                    if(q->bf!=0){ //neu bi mat can bang
                            fya=p;
```

```
            ya=q;
                }
            fp=p;
            p=q;
    }
    q=makenode(x);
   //them vao mot node la con cua fp
    if(x<fp->info)      fp->left=q;
    else                fp->right=q;
   //hieu chinh lai chi so can bang cua tat ca cac node giua ya va q
    if(x<ya->info)              p=ya->left;
    else                p=ya->right;

    s=p;
    while(p!=q){
            if(x<p->info){
                    p->bf=1;
                    p=p->left;
            }else{
                    p->bf=-1;
                p=p->right;
            }
    }

    //xac dinh huong lech
    if(x<ya->info)      imbal=1;
    else                imbal=-1;

    if(ya->bf==0){
            ya->bf=imbal;
      return;
    }
    if(ya->bf !=imbal){
            ya->bf=0;
      return;
    }

    if(s->bf==imbal){
            if(imbal==1){
            p=RotateRight(ya);
            }else{
            p=RotateLeft(ya);
            }
```

```
            ya->bf=0;
        s->bf=0;
    }else{
            if(imbal==1){
                    ya->left=RotateLeft(s);
        p=RotateRight(ya);
            }else{
                    ya->right=RotateRight(s);
        p=RotateLeft(ya);
            }
            if(p->bf==0){
                    ya->bf=0;
        s->bf=0;
            }else
                    if(p->bf==imbal){
                            ya->bf=-imbal;
                            s->bf=0;
                    }else{
                            ya->bf=0;
             s->bf=imbal;
                    }
            p->bf=0;
    }
    if(fya==NULL)       *pavltree=p;
    else
            if(ya==fya->right)   fya->right=p;
            else            fya->left=p;
}
```

TÓM TẮT

Trong bài này, học viên cần nắm:

Cây AVL (do tác giả Adelson-Velskii và Landis khám phá) là "cây nhị phân cân bằng": Là cây nhị phân tìm kiếm; Tại mỗi nút: Độ cao của cây con trái và cây con phải chênh lệch ko quá 1.

Các thao tác trên cây AVL tương tự như BST, khác biệt khi thêm/xoá sẽ làm mất cân bằng: Ảnh hưởng đến chỉ số cân bằng của nhánh cây liên quan, Sử dụng thao tác xoay phải, trái để cân bằng.

CÂU HỎI ÔN TẬP

Câu 1: Xét tác vụ insert và remove để thêm nút và xoá nút trên cây AVL.

a. Vẽ lại hình ảnh của cây AVL nếu thêm các nút vào cây theo thứ tự như sau: 8, 3, 5, 2, 20, 11, 30, 9, 18, 4.

b. Vẽ lại hình ảnh của cây trên nếu ta lần lược xoá 2 nút 5 và 20.

Câu 2: Cài đặt cấu trúc dữ liệu liên kết cho cây AVL, với các thao tác:

a. Cài đặt các thao tác xây dựng cây: Init, IsEmpty, CreateNode

b. Cài đặt thao tác cập nhật: Insert, Remove

BÀI 6: ĐỒ THỊ VÀ BIỂU DIỄN ĐỒ THỊ

Sau khi học xong bài này, học viên có thể:

- *Hiểu được các khái niệm về đồ thị, bao gồm: đồ thị vô hướng, đồ thị có hướng,*
- *Cách biểu diễn đồ thị.*
- *Ứng dụng của đồ thị vào các bài toán thực tế.*

6.1 CÁC KHÁI NIỆM

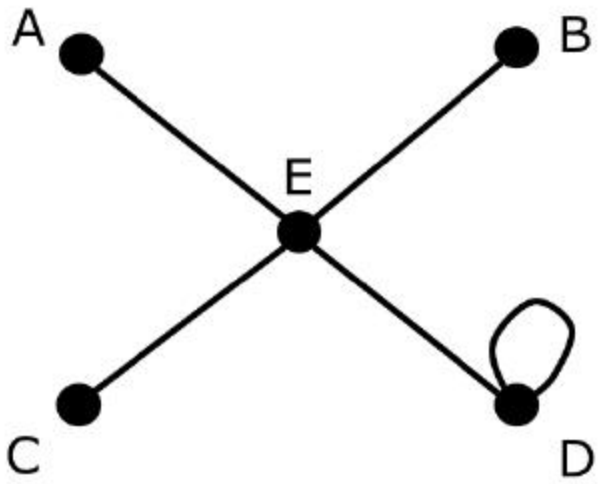
6.1.1 Đồ thị

Đồ thị G = (V, E) gồm một tập V gọi là tập đỉnh và một tập E gọi là tập cạnh hay cung. Tập $E \subseteq V^2$ gồm các cặp phần tử của V. Giả sử u và v là hai đỉnh của đồ thị G (u, v ∈ V), nếu cặp đỉnh (u, v) không được sắp thứ tự thì (u, v) gọi là cạnh nối hai đỉnh u và v. Ngược lại, nếu cặp đỉnh (u, v) được sắp thứ tự thì (u, v) gọi là cạnh có hướng (hay cung), trong đó u được gọi là đỉnh đầu và v được gọi là đỉnh cuối.

Đồ thị vô hướng là đồ thị chỉ chứa các cạnh trong đồ thị vô hướng, cạnh (u, v) tương đương với cạnh (v, u).

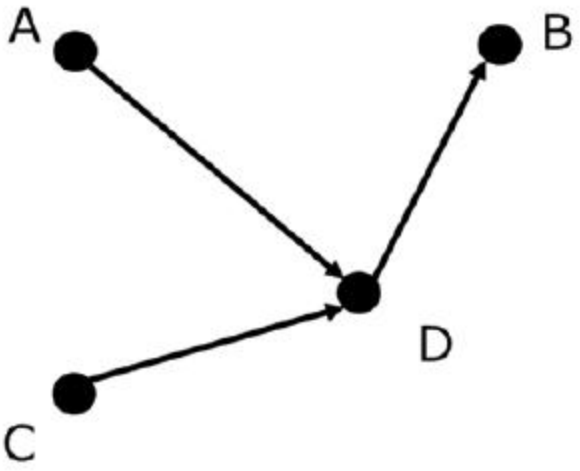
Đồ thị có hướng là đồ thị chỉ chứa các cạnh có hướng (cung). Trong đồ thị có hướng, cung (u, v) khác với cung (v, u).

Về mặt hình học, mỗi đỉnh trong đồ thị vô hướng được biểu diễn bởi một điểm, mỗi cạnh được biểu diễn bởi đường nối giữa hai điểm. Hình 6.1 biểu diễn đồ thị vô hướng có tập đỉnh gồm 5 đỉnh {A, B, C, D, E}, và tập cạnh gồm 5 cạnh {(A, E), (B, E), (C, E), (D, E), (D, D)}.



Hình 6.1: Ví dụ về đồ thị vô hướng

Trong đồ thị có hướng, đỉnh cũng được biểu diễn bởi một điểm, tuy nhiên, mỗi cạnh được biểu diễn bởi một đường có hướng (mũi tên) từ đỉnh đầu sang đỉnh cuối. Hình 6.2 biểu diễn đồ thị có hướng có tập đỉnh gồm 3 đỉnh {A, B, C}, và tập cung gồm 3 cung {(A, D), (D, B), (C, D)}.



Hình 6.2: Ví dụ về đồ thị có hướng

Một số khái niệm trên đồ thị:

Khuyên: Cạnh nối một đỉnh với chính nó được gọi là một khuyên. Trong đồ thị trên hình 6.1, (D, D) là một khuyên.

Đỉnh kề và cạnh liên thuộc: Trong đồ thị G = (V, E), hai đỉnh u, v ∈ V (u ≠ v) được gọi là kề nhau nếu tồn tại cạnh e = (u, v) ∈ E. Khi đó, cạnh e được gọi là liên thuộc với đỉnh u và v.

Đồ thị trong hình 6.1 có các cặp đỉnh kề sau: A và E, B và E, C và E, D và E. (A, E) là cạnh liên thuộc với đỉnh A và E, (B, E) là cạnh liên thuộc với đỉnh B và E, (C, E) là cạnh liên thuộc với đỉnh C và E, (D, E) là cạnh liên thuộc với đỉnh D và E.

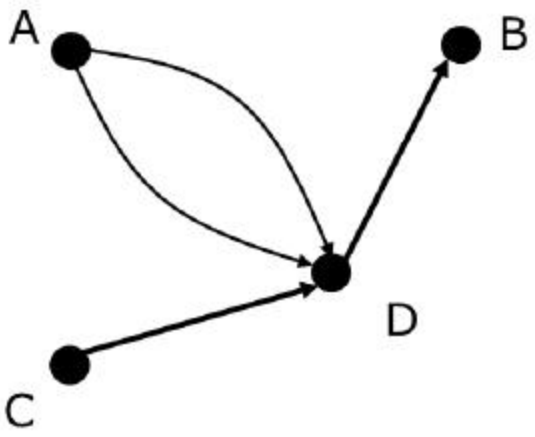
Đồ thị trong hình 6.2 có các cặp đỉnh kề nhau sau: A và D, B và D, C và D. (A, D) là cung liên thuộc với đỉnh A và D, (B, D) là cung liên thuộc với đỉnh B và D, (C, D) là cung liên thuộc với đỉnh C và D.

Cạnh (cung) kề nhau: Hai cung e1 và e2 (e1 ≠ e2) được gọi là kề nhau nếu chúng có đỉnh chung (nếu e1 và e2 là cung thì không phụ thuộc vào việc đỉnh chung đó là đỉnh đầu hay đỉnh cuối của cung e1, đỉnh đầu hay đỉnh cuối của cung e2).

Đồ thị trong hình 6.1 có các cặp cạnh kề nhau sau: (A, E) và (B, E), (A, E) và (C, E), (A, E) và (D, E), (B, E) và (C, E), (B, E) và (D, E), (C, E) và (D, E).

Đồ thị trong hình 6.2 có các cặp cạnh kề nhau sau: (A, D) và (B, D), (A, D) và (C, D), (B, D) và (C, D).

Cạnh (cung) song song: Hai cạnh (cung) được gọi là song song nếu nó nối hai cặp đỉnh giống nhau. Đồ thị trong hình 6.3 có cung song song nối giữa hai đỉnh A và D.

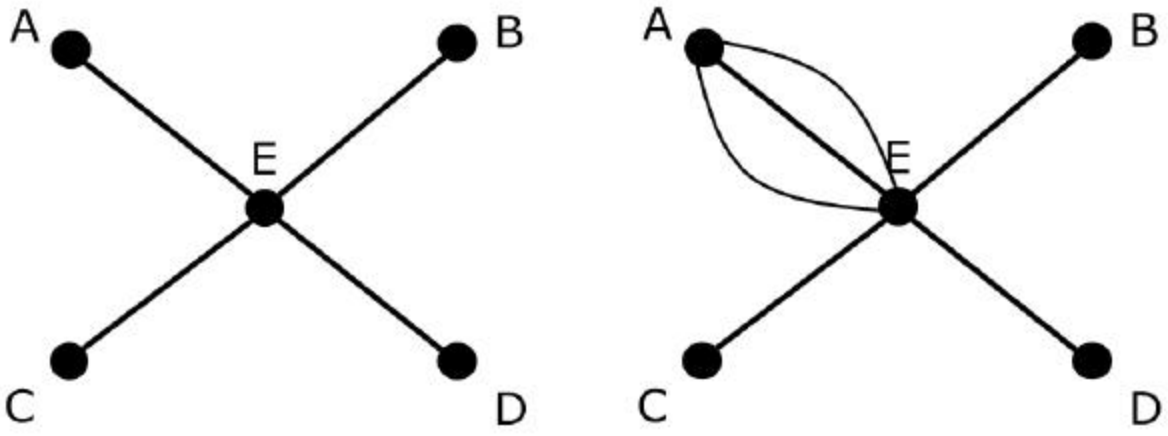


Hình 6.3: Ví dụ về cung song song

6.1.2 Một số dạng đồ thị đặc biệt

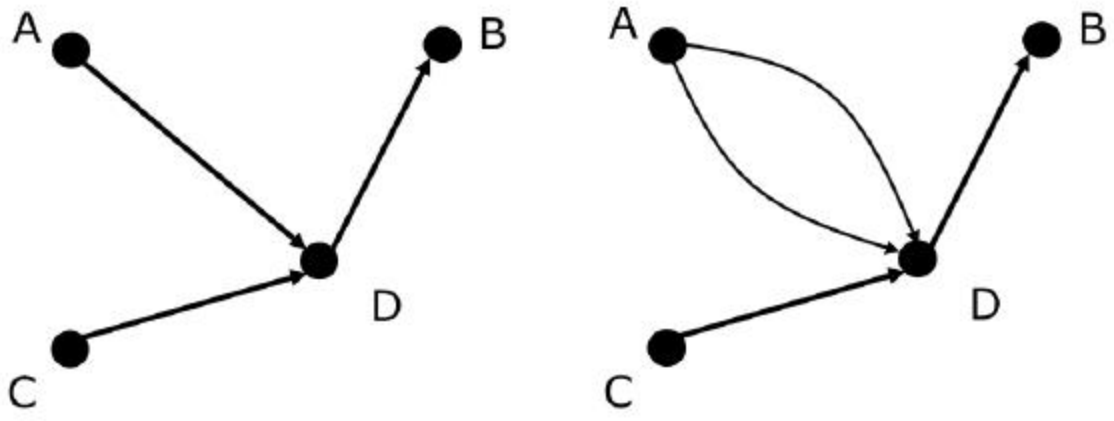
Đơn đồ thị: là đồ thị không chứa khuyên và mỗi cặp đỉnh chỉ được nối bởi một cạnh duy nhất.

Đa đồ thị: là đồ thị mà mỗi cặp đỉnh có thể được nối bởi nhiều hơn một cạnh. Ví dụ về đơn đồ thị vô hướng và đa đồ thị vô hướng được biểu diễn trong Hình 6.4 – Hình 6.5.



(a) (b)

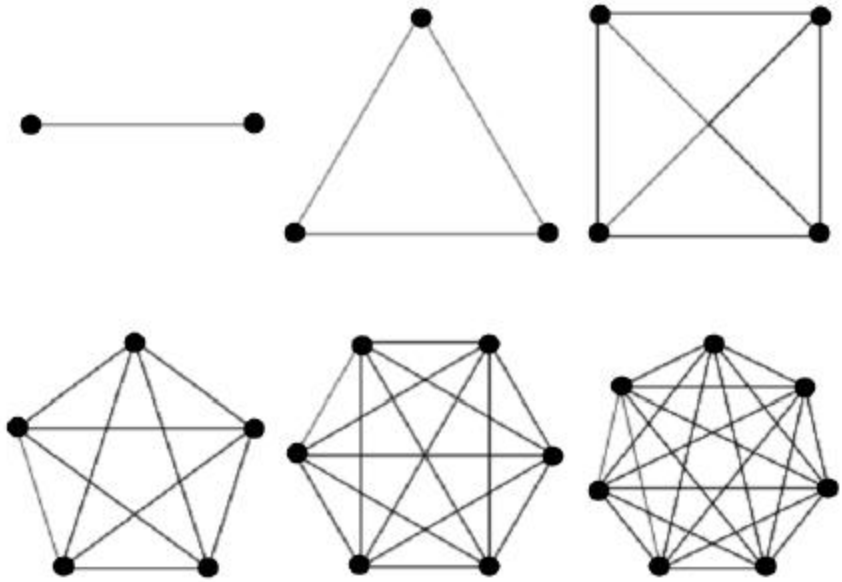
Hình 6.4: (a) Đơn đồ thị vô hướng, (b) Đa đồ thị vô hướng



(a) (b)

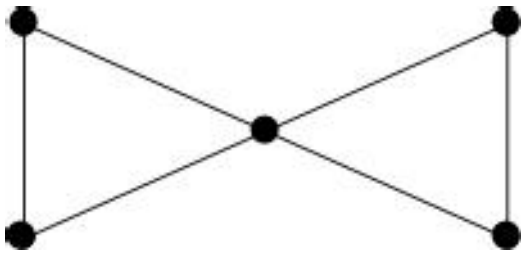
Hình 6.5: (a) Đơn đồ thị có hướng, (b) đa đồ thị có hướng

Đồ thị đầy đủ: Đồ thị G = (V, E) được gọi là đồ thị đầy đủ nếu mỗi cặp đỉnh được nối với nhau bằng đúng một cạnh (cung). Hình 6.6 là một số ví dụ về đồ thị đầy đủ.

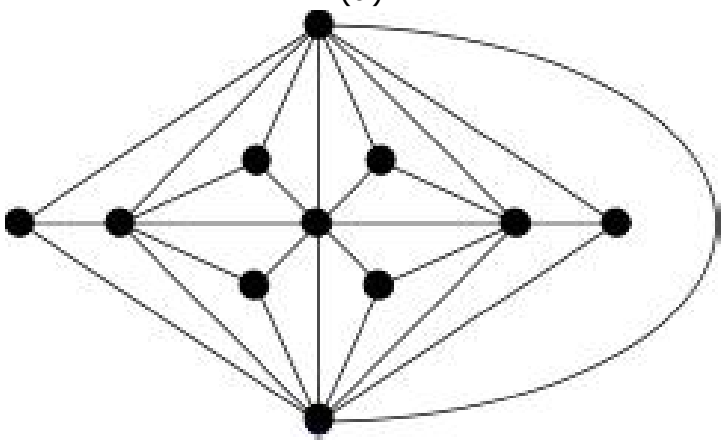


Hình 6.6: Ví dụ về đồ thị đầy đủ

Đồ thị phẳng: đồ thị G = (V, E) được gọi là đồ thị phẳng nếu có thể biểu diễn hình học đồ thị G trên một mặt phẳng nào đó mà các cạnh của đồ thị chỉ cắt nhau ở đỉnh.



(a)



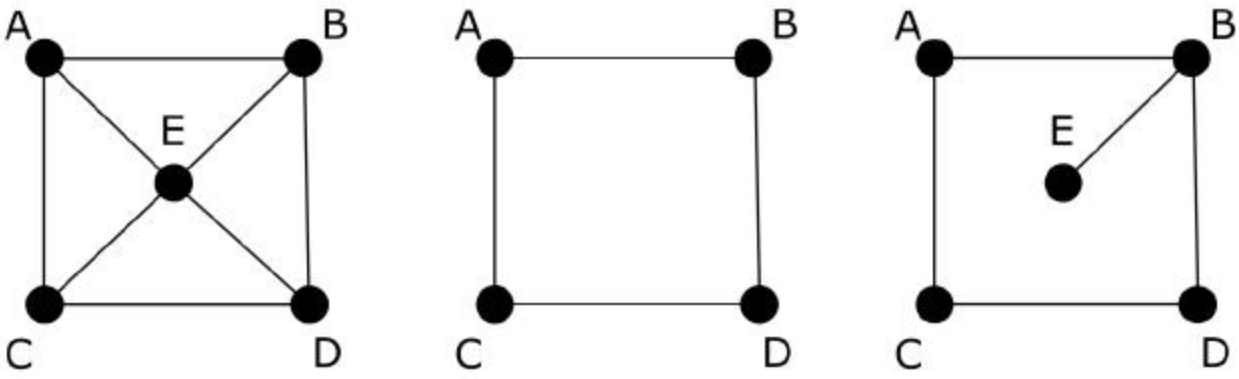
(b)

Hình 6.7: Ví dụ về đồ thị phẳng:
(a) Đồ thị cánh bướm (đồ thị hình nơ) (b) Đồ thị Goldner–**Harary**

Đồ thị con: Đồ thị $G_A = (V_A, E_A)$ được gọi là đồ thị con của đồ thị $G = (V_G, E_G)$ nếu V_A là tập con của V_G và các cung trong E_A là các cạnh/cung của G mà hai đỉnh nó liên thuộc thuộc tập V_A.

Đồ thị bộ phận: Đồ thị $G_1 = (V_G, E_1)$ là đồ thị bộ phân của đồ thị $G = (V_G, E_G)$ nếu E_1 là tập con của E_G.

Hình 6.8 là ví dụ về đồ thị con và đồ thị bộ phận. Đồ thị G trong hình 6.8.a có tập đỉnh V_G = {A, B, C, D, E} và tập cạnh E_G = {(A, B), (A, C), (A, E), (B, D), (B, E), (C, D), (C, E), (D, E)}. Đồ thị G_1 trong hình 6.8.b có tập đỉnh V_{G1} = {A, B, C, D} và tập cạnh E_{G1} = {(A, B), (A, C), (B, D), (C, D)}. Do $V_{G1} \subset V_G$, đồ thị G1 là đồ thị con của đồ thị G. Đồ thị G2 trong hình 6.8.c có tập đỉnh V_{G2} = {A, B, C, D, E} và tập cạnh E_{G2} = {(A, B), (A, C), (B, D), (B, E), (C, D)}. Do $V_G = V_{G2}$ và $E_{G2} \subset E_G$, đồ thị G2 là đồ thị bộ phận của đồ thị G.



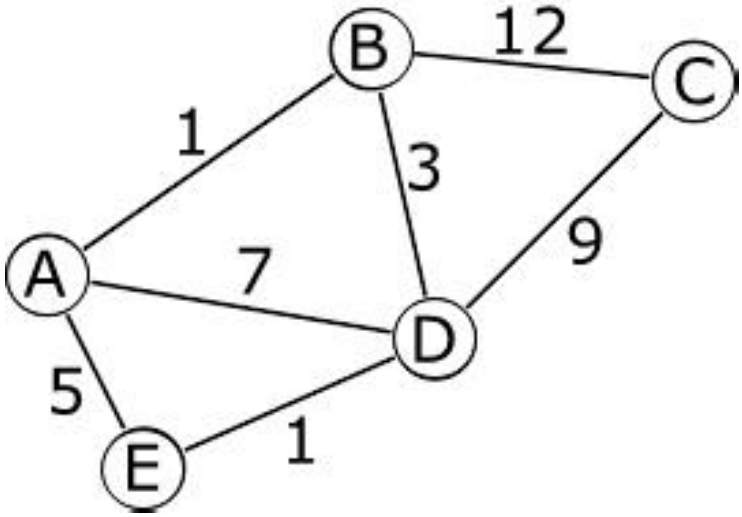
(a) (b) (c)

Hình 6.8: Ví dụ về đồ thị con và đồ thị bộ phận

Đồ thị bộ phận con: là đồ thị vừa là đồ thị con vừa là đồ thị bộ phận.

Đồ thị có trọng số: Đồ thị có trọng số là đồ thị mà mỗi cạnh (cung) (u, v) có một giá trị c(u, v) gọi là trọng số của cạnh. Về mặt hình học, trọng số của mỗi cạnh (cung) được ghi trên mỗi cạnh (cung). Hình 6.9 là ví dụ về đồ thị có trọng số, trong đó giá trị trọng số của mỗi cạnh được thể hiện trong bảng sau:

Cạnh	Trọng số
(A, B)	1
(A, D)	7
(A, E)	5
(B, C)	12
(B, D)	3
(C, D)	9
(D, E)	1



Hình 6.9: Ví dụ về đồ thị có trọng số

6.1.3 Bậc của đỉnh đồ thị

Bậc của đỉnh: Trong đồ thị vô hướng (hoặc có hướng), bậc của đỉnh v là số cạnh liên thuộc với đỉnh v.

Đồ thị trong hình 6.9 có bậc của các đỉnh lần lượt như sau: bậc(A) = 3, bậc(B) = 3, bậc(C) = 2, bậc(D) = 4, bậc(E) = 2.

Đỉnh treo và đỉnh cô lập: Nếu bậc của đỉnh bằng 1, đỉnh được gọi là đỉnh treo. Nếu bậc của đỉnh bằng 0, đỉnh được gọi là đỉnh cô lập.

Cạnh (cung) treo: là cạnh (cung) có ít nhất một đầu là đỉnh treo.

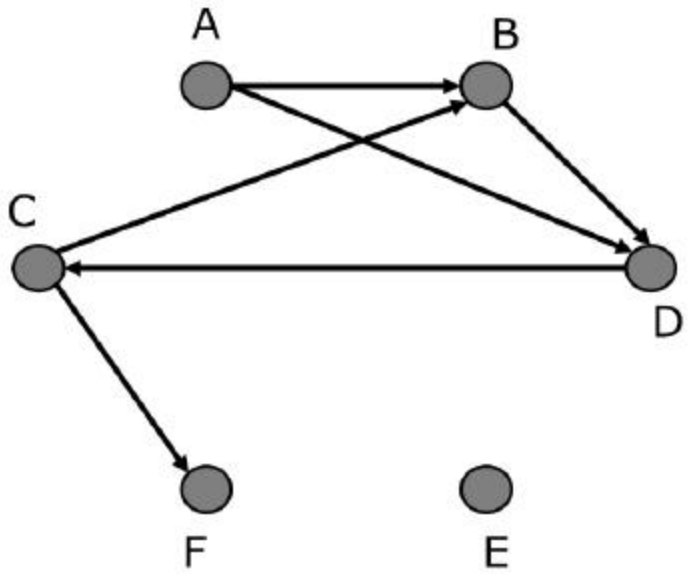
Nửa bậc trong: nửa bậc trong của đỉnh v, ký hiệu $d^-(v)$ là số cung có v là đỉnh cuối (tương ứng với số cung đi vào v).

Nửa bậc ngoài: nửa bậc ngoài của đỉnh v, ký hiệu $d^+(v)$ là số cung có v là đỉnh đầu (tương ứng với số cung đi ra từ v).

Trong đồ thị có hướng, bậc của đỉnh v, ký hiệu d(v), bằng $d^-(v) + d^+(v)$.

Đồ thị trong Hình 6.10 có nửa bậc trong, nửa bậc ngoài, và bậc của các đỉnh thể hiện trong bảng bên dưới. Đỉnh E của đồ thị đã cho được gọi là đỉnh cô lập, đỉnh F được gọi là đỉnh treo. Cung (C, F) được gọi là cung treo.

Đỉnh	A	B	C	D	E	F
Nửa bậc trong	0	2	1	2	0	1
Nửa bậc ngoài	2	1	2	1	0	0
Bậc	2	3	3	3	0	1



Hình 6.10: Ví dụ về bậc của đồ thị có hướng

Một số tính chất về bậc của đỉnh đồ thị:

1. Tổng số bậc của tất cả các đỉnh gấp đôi số cạnh.
2. Số đỉnh có bậc lẻ luôn là một số chẵn
3. Nếu đồ thị có nhiều hơn hai đỉnh thì có ít nhất hai đỉnh cùng bậc.
4. Nếu một đồ thị với n đỉnh (n > 2) có đúng hai đỉnh cùng bậc thì hai đỉnh này không thể có bậc 0 hoặc n – 1.
5. Luôn tồn tại đồ thị n đỉnh (n > 2) mà 3 đỉnh bất kỳ của đồ thị đều không cùng bậc.

6. Cho đồ thị G = (V, E) với ít nhất kn + 1 đỉnh, mỗi đỉnh có bậc không bé hơn (k – 1) n + 1, luôn tồn tại đồ thị con đầy đủ của G gồm k + 1 đỉnh.

6.1.4 Đường đi và chu trình

Giả sử G = (V, E) là một đồ thị vô hướng (hoặc có hướng).

Đường đi: dãy $v_0, v_1, ..., v_n$ ($v_i \in V$, i = 0, 1, ..., n) được gọi là đường đi từ v_0 đến v_n nếu $\forall$ i (1 ≤ i ≥ n – 1) cặp đỉnh v_i và v_{i+1} kề nhau, nghĩa là $(v_i, v_{i+1}) \in E$. Khi đó, v_0 được gọi là đỉnh bắt đầu của đường đi và v_n được gọi là đỉnh kết thúc của đường đi. Độ dài của đường đi là số cạnh xuất hiện trong dãy $v_0, v_1, ..., v_n$.

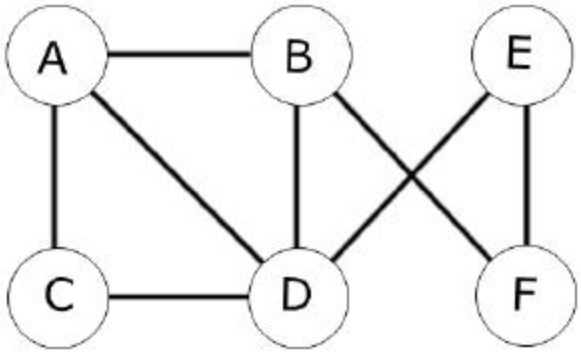
Đường đi sơ cấp: là đường đi mà các đỉnh trong đường đi không bị lặp lại.

Chu trình: là đường đi có đỉnh bắt đầu và đỉnh kết thúc trùng nhau.

Chu trình sơ cấp: là chu trình có các đỉnh không bị lặp lại, trừ đỉnh đầu và đỉnh cuối.

Cho đồ thị vô hướng như trong hình 6.11, dãy {A, B, F, E} là một đường đi từ A đến E của đồ thị. Tuy nhiên, dãy {A, B, E, F} không phải là đường đi của đồ thị. Đường đi {A, B, F, E} là đường đi sơ cấp. đường đi {A, B, D, E, F, B} không phải là đường đi sơ cấp.

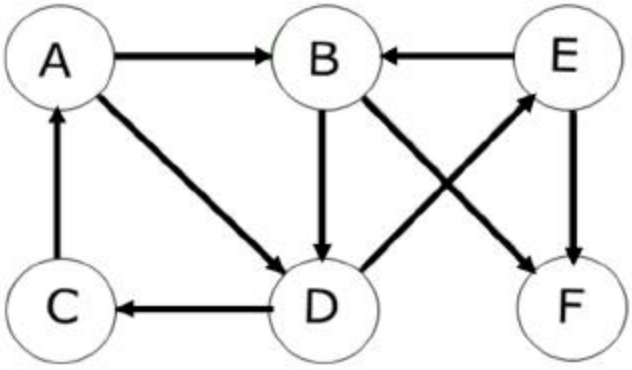
Đồ thị trong hình 6.1 có một số chu trình như sau: {A, B, D, A}, {A, C, D, A}, {A, B, F, E, D, A}, {A, B, F, E, D, C, A}, {A, D, B, F, E, D, C, A}. Trong số đó, {A, D, B, F, E, D, C, A} không phải là chu trình sơ cấp, còn lại các chu trình khác đều là chu trình sơ cấp.



Hình 6.11: Ví dụ về đường đi và chu trình trên đồ thị vô hướng

Cho đồ thị có hướng trong hình 6.12, một số đường đi từ đỉnh A đến F như sau: {A, B, F}, {A, B, D, E, F}, {A, D, E, F}, {A, B, D, E, B, F}. Trong số các đường đi từ A đến F vừa liệt kê ở trên, {A, B, F}, {A, B, D, E, F}, {A, D, E, F} là đường sơ cấp, còn {A, B, D, E, B, F} không phải là đường sơ cấp. {A, D, B, F} không phải là một đường đi từ A đến F.

Đồ thị trong hình 6.12 có một số chu trình như sau: {A, D, C, A}, {A, B, D, C, A}.



Hình 6.12: Ví dụ về đường đi và chu trình trên đồ thị có hướng

Một số tính chất:

1. Giả sử G là đồ thị vô hướng với n đỉnh (n > 2) và các đỉnh đều có bậc không nhỏ hơn 2. Khi đó, G chứa ít nhất một chu trình sơ cấp.
2. Giả sử G là đồ thị vô hướng với n đỉnh (n > 3) và các đỉnh đều có bậc không nhỏ hơn 3. Khi đó, G chứa ít nhất một chu trình sơ cấp có độ dài chẵn.

6.2 TÍNH LIÊN THÔNG CỦA ĐỒ THỊ

6.2.1 Định nghĩa

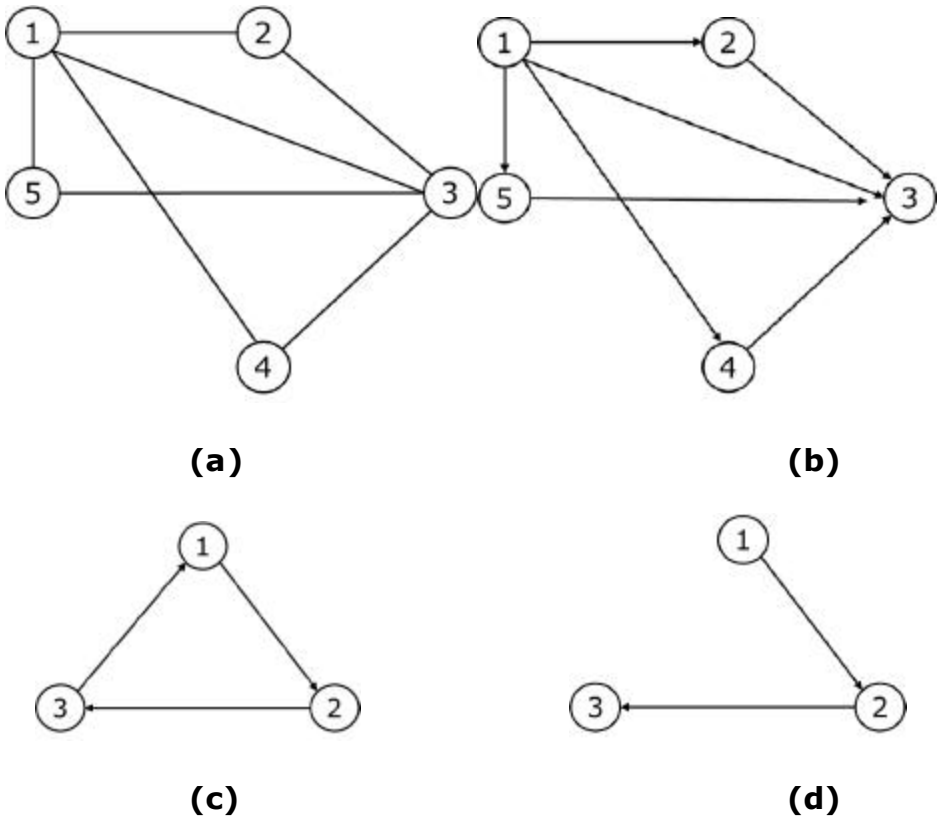
Cặp đỉnh liên thông: Hai đỉnh u, v được gọi là cặp đỉnh liên thông nếu tồn tại ít nhất một đường đi từ u sang v hoặc từ v sang u.

Đồ thị liên thông: đồ thị vô hướng G = (V, E) được gọi là đồ thị liên thông nếu với mỗi cặp đỉnh bất kỳ u, v thì luôn tìm được đường đi từ u đến v, nghĩa là mọi cặp đỉnh của G đều liên thông.

Đồ thị có hướng liên thông mạnh: đồ thị có hướng G = (V, E) là liên thông mạnh nếu với hai đỉnh bất kỳ u và v, tồn tại đường đi từ u đến v, và ngược lại.

Đồ thị có hướng liên thông yếu: đồ thị có hướng G = (V, E) là liên thông yếu nếu đồ thị vô hướng tương ứng với G là liên thông.

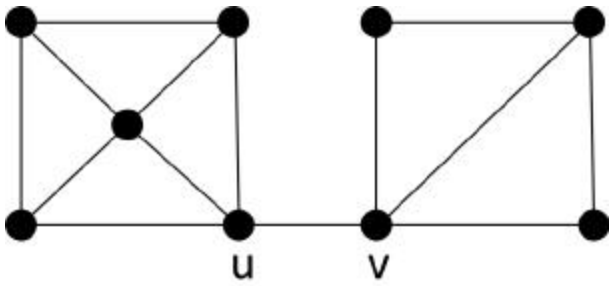
Đồ thị có hướng liên thông một phần: đồ thị có hướng G = (V, E) là liên thông một phần nếu với hai đỉnh bất kỳ u và v, hoặc là tồn tại đường đi từ u đến v, hoặc là tồn tại đường đi từ v đến u.



Hình 6.13: (a) Đồ thị vô hướng liên thông,
(b) Đồ thị có hướng liên thông yếu, (c) Đồ thị có hướng liên thông mạnh,
(d) Đồ thị có hướng liên thông một phần

Đỉnh khớp: đỉnh khớp của một đồ thị vô hướng là đỉnh mà nếu xóa khỏi đồ thị đỉnh đó và các cạnh nối đến nó thì số thành phần liên thông của đồ thị sẽ tăng lên.

Cạnh cầu: cạnh của một đồ thị vô hướng được gọi là cạnh cầu nếu xóa cạnh đó khỏi đồ thị thì số thành phần liên thông của đồ thị sẽ tăng thêm.



Hình 6.14: Ví dụ về đỉnh khớp và cạnh cầu: Đồ thị có 2 đỉnh khớp là u và v, có một cạnh cầu là (u, v)

6.2.2 Các tính chất

Tính liên thông của đồ thị có một số tính chất sau:

1. Đồ thị với n đỉnh ($n \geq 2$) có tổng bậc của hai đỉnh bất kỳ lớn hơn hoặc bằng n là đồ thị liên thông.
2. Đồ thị có bậc của mỗi đỉnh không nhỏ hơn một nửa số đỉnh là đồ thị liên thông.
3. Nếu đồ thị có đúng hai đỉnh bậc lẻ thì hai đỉnh này là hai đỉnh liên thông.
4. Đồ thị G liên thông khi và chỉ khi nó có một thành phần liên thông duy nhất.
5. Nếu đồ thị G có n đỉnh và số cạnh lớn hơn (n-1)(n-2)/2 thì đồ thị G là liên thông.

6.3 BIỂU DIỄN ĐỒ THỊ

Để lưu trữ và thực hiện các thuật toán trên đồ thị, chúng ta cần lựa chọn cấu trúc dữ liệu phù hợp để biểu diễn đồ thị. Cách thức biểu diễn đồ thì ảnh hướng tới hiệu quả của thuật toán, do đó, việc lựa chọn cấu trúc dữ liệu nào tùy thuộc vào bài toán cụ thể. Phần này sẽ giới thiệu một số phương pháp biểu diễn đồ thị cơ bản, bao gồm: ma trận kề, ma trận trọng số, danh sách cạnh, và danh sách kề.

6.3.1 Ma trận kề

Xét đồ thị vô hướng G = (V, E) với tập đỉnh V = {v_1, v_2, …, v_n} và tập cạnh E = {e_1, e_2, …, e_m}. Ma trận kề của đồ thị G là ma trận vuông A cấp $n \times n$ với các phần tử được xác định như sau:

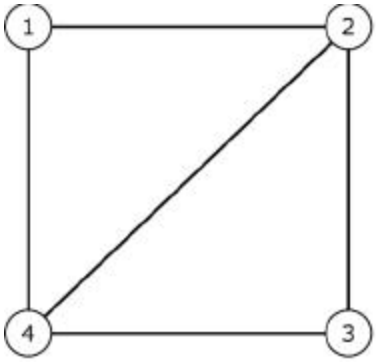
$$A_{ij} = \begin{cases} 1 \text{ nếu } (v_i, v_j) \in E \\ 0 \quad \text{nếu ngược lại} \end{cases}$$

Trong trường hợp đồ thị vô hướng G là đa đồ thị, mỗi phần tử A_{ij} của ma trận kề của đồ thị được xác định bằng số lượng cạnh nối giữa hai đỉnh v_i và v_j.

Ma trận kề của đồ thị vô hướng có một số tính chất sau:

1. Ma trận kề là ma trận đối xứng. Do vậy để giảm không gian bộ nhớ lưu trữ đồ thị, đối với đồ thị vô hướng ta có thể chỉ cần lưu trữ nửa tam giác trên của ma trận kề. Tuy nhiên, cách lưu trữ này có thể làm tăng độ phức tạp trong một số bài toán.
2. Tổng các phần tử trên dòng i của ma trận kề bằng bậc của đỉnh i.
3. Tổng các phần tử trên cột j của ma trận kề bằng bậc của đỉnh j.

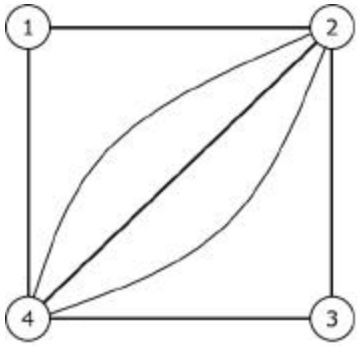
Ví dụ 6.1: ma trận kề của đơn đồ thị vô hướng.



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Hình 6.15: Đồ thị vô hướng và ma trận kề tương ứng

Ví dụ 6.2: ma trận kề của đa đồ thị vô hướng.

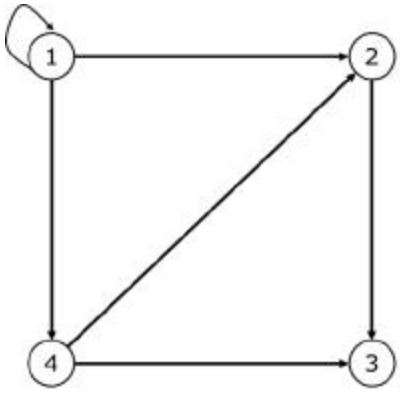


$$A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 3 \\ 0 & 1 & 0 & 1 \\ 1 & 3 & 1 & 0 \end{bmatrix}$$

Hình 6.16: Đa đồ thị vô hướng và ma trận kề tương ứng

Ma trận kề của đồ thị có hướng được định nghĩa hoàn toàn tương tự như đối với đồ thị vô hướng. Tuy nhiên, ma trận kề của đồ thị có hướng không có tính chất đối xứng.

Ví dụ 6.3: ma trận kề của đơn đồ thị có hướng.



$$A = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Hình 6.17: Đồ thị có hướng và ma trận kề tương ứng

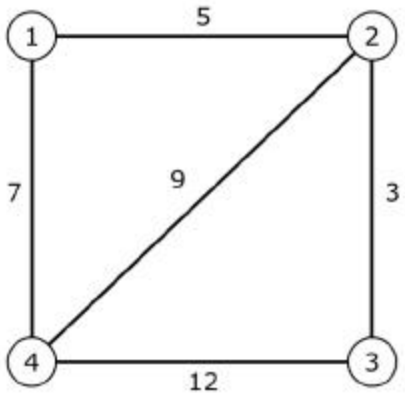
6.3.2 Ma trận trọng số

Xét đồ thị G = (V, E) là đồ thị có trọng số với tập đỉnh V = {v_1, v_2, …, v_n} và tập cạnh E = {e_1, e_2, …, e_m}. Mỗi cạnh (v_i, v_j) có trọng số là c_{ij}. Tương tự như ma trận kề, ma trận trọng số của đồ thị G là ma trận vuông cấp $n \times n$ với các phần tử được xác định bằng trọng số của mỗi cạnh như sau:

$$A_{ij} = \begin{cases} c_{ij} \text{ nếu } (v_i, v_j) \in E \\ 0 \quad \text{nếu ngược lại} \end{cases}$$

Ma trận trọng số của đồ thị vô hướng cũng là ma trận đối xứng. Tuy nhiên, tổng các phần tử trên dòng i của ma trận trọng số không bằng bậc của đỉnh i.

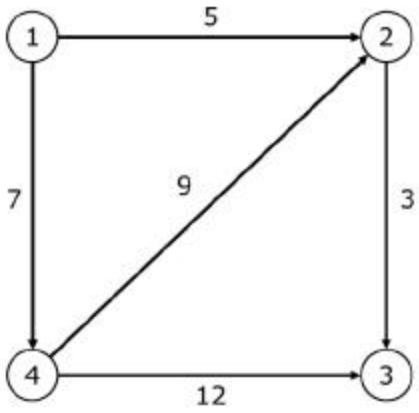
Ví dụ 6.4: ma trận trọng số của đồ thị vô hướng.



$$A = \begin{bmatrix} 0 & 5 & 0 & 7 \\ 5 & 0 & 3 & 9 \\ 0 & 3 & 0 & 12 \\ 7 & 9 & 12 & 0 \end{bmatrix}$$

Hình 6.18: Đồ thị vô hướng có trọng số và ma trận trọng số tương ứng

Ví dụ 6.5: ma trận trọng số của đồ thị có hướng.



$$A = \begin{bmatrix} 0 & 5 & 0 & 7 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 9 & 12 & 0 \end{bmatrix}$$

Hình 6.19: Đồ thị có hướng có trọng số và ma trận trọng số tương ứng

6.3.3 Danh sách cạnh (cung)

Phương pháp biểu diễn đồ thị bằng ma trận kề đặc biệt lãng phí bộ nhớ trong trường hợp đồ thị thưa (nghĩa là đồ thị có ít cạnh so với số đỉnh). Trong trường hợp này, người ta thường sử dụng cách biểu diễn đồ thị theo danh sách cạnh.

Để biểu diễn đồ thị bằng phương pháp danh sách cạnh (cung), duy trì hai danh sách, mỗi danh sách gồm m phần tử. Trong đó, phần tử tại vị trí i của danh sách thứ nhất chứ đỉnh đầu của cạnh e_i, và phần tử tại vị trí i của danh sách thứ hai chứa đỉnh cuối của cạnh e_i (i = 1, ..., m). Trong trường hợp đồ thị có trọng số, duy trì thêm một danh sách thứ ba để lưu trữ trọng số của cạnh e_i.

Ví dụ 6.6: danh sách cung của đồ thị có hướng **trong hình 6.17 là:**

Đỉnh đầu	Đỉnh cuối
1	1
1	2
1	4
2	3
4	2
4	3

Ví dụ **6.7**: danh sách cạnh của đồ thị có hướng có trọng số trong hình 6.19 là:

Đỉnh đầu	Đỉnh cuối	Trọng số
1	2	5
1	4	7
2	3	3
4	2	9
4	3	12

6.3.4 Danh sách kề

Trong phương pháp biểu diễn đồ thị bằng danh sách kề, với mỗi đỉnh u của đồ thị G = (V, E), duy trì một danh sách các đỉnh kề với u, ký hiệu L(u). Khi đó, L(u) = {v ∈ V: (u, v) ∈ E).

Ví dụ **6.8**: danh sách kề của đồ thị vô hướng trong hình 6.15.

L(1) = {2, 4}

L(2) = {1, 3, 4}

L(3) = {2, 4}

L(4) = {1, 2, 3}

Ví dụ **6.9**: danh sách cạnh của đồ thị có hướng trong hình 6.17.

L(1) = {1, 2, 4}

L(2) = {3}

L(3) = ∅

L(4) = {2, 3}

TÓM TẮT

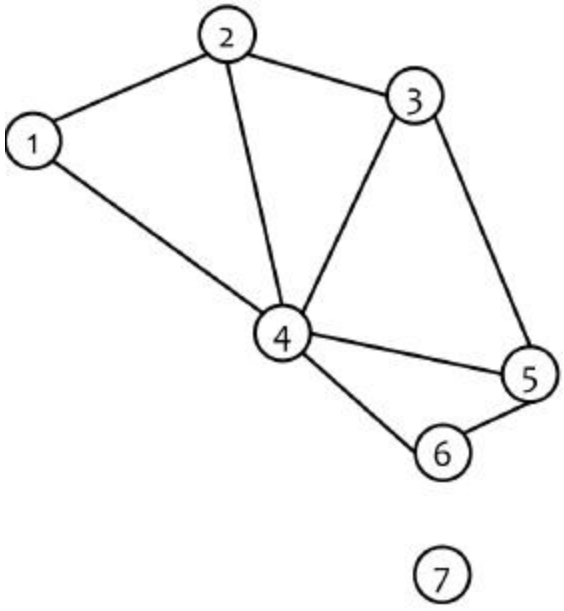
Trong bài này, học viên cần nắm:

Khái niệm về đồ thị vô hướng, đồ thị có hướng, đồ thị có trọng số và một số dạng đồ thị đặc biệt khác. Học viên cũng cần nắm được các khái niệm cơ bản khác trong đồ thị như đỉnh kề, cạnh (cung) kề, bậc của đỉnh trong đồ thị vô hướng và đồ thị có hướng, đường đi, chu trình, tính liên thông của đồ thị.

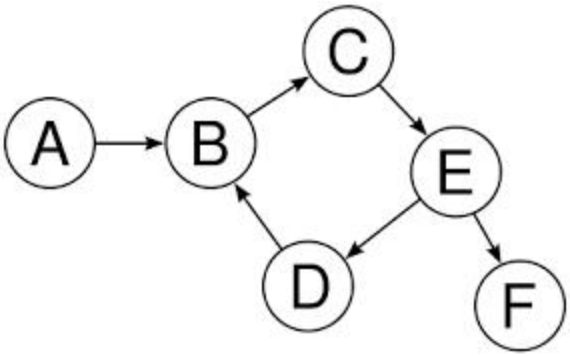
Đồ thị có thể được biểu diễn bằng các phương pháp sau: ma trận kề, ma trận trọng số, danh sách cạnh (cung), và danh sách kề. Các cách biểu diễn đồ thị ảnh hưởng tới hiệu quả của các thuật toán trên đồ thị sẽ học sau này. Do đó, học viên cần nắm được cách thức biểu diễn, ưu và nhược điểm của mỗi phương pháp biểu diễn đồ thị. Từ đó, lựa chọn được cách biểu diễn đồ thị phù hợp cho các bài toán ứng dụng.

CÂU HỎI ÔN TẬP

Câu 1: Hãy biểu diễn đồ thị sau sử dụng ma trận kề, danh sách cạnh (cung), và danh sách kề.



Câu 2: Hãy biểu diễn đồ thị sau sử dụng ma trận trọng số, danh sách cạnh (cung), và danh sách kề.



Câu 3: Hãy vẽ đồ thị tương ứng với ma trận kề sau:

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Câu 4: Hãy vẽ đồ thị tương ứng với ma trận trọng số sau:

$$\begin{bmatrix} 0 & 1 & 1 & 2 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

Câu 5: Xác định một đồ thị con, một đồ thị bộ phận của đồ thị xây dựng trong câu 3 và câu 4.

Câu 6: Xác định các thành phần liên thông (hoặc liên thông mạnh) của đồ thị xây dựng trong câu 3 và câu 4.

Câu 7: Xây dựng thuật toán biểu diễn đồ thị bằng ma trận kề, ma trận trọng số, danh sách cạnh (cung), và danh sách kề.

Câu 8: Xây dựng thuật toán tính bậc của các đỉnh của một đồ thị khi biết ma trận kề của đồ thị.

BÀI 7: ĐƯỜNG ĐI VÀ CHU TRÌNH

Sau khi học xong bài này, học viên có thể:

- *Hiểu được chu trình Euler.*
- *Hiểu được chu trình Hamilton.*
- *Vận dụng chu trình Euler và Hamilton để giải các bài toán cụ thể.*

7.1 CHU TRÌNH EULER

7.1.1 Định nghĩa

Cho đồ thị G = (V, E) có n đỉnh, với V = {v_1, v_2, …, v_n }, E = {e_1, e_2, …, e_m}

Đường đi Euler: đường đi qua tất cả các cạnh của đồ thị mỗi cạnh đúng một lần, trong đó đỉnh đầu và đỉnh cuối khác nhau.

Điều kiện cần và đủ để tồn tại đường đi Euler

- Đồ thị vô hướng:
  - Đồ thị liên thông
  - Tồn tại duy nhất 2 đỉnh bậc lẻ, tất cả các đỉnh còn lại bậc chẵn
  - Phải xuất phát từ đỉnh bậc lẻ thứ nhất và kết thúc là đỉnh bậc lẻ còn lại
- Đồ thị có hướng
  - Đồ thị liên thông
  - Các đỉnh có bậc ngoài bằng bậc trong, trừ 1 đỉnh có bậc ngoài lớn hơn bậc trong 1 đơn vị (đỉnh xuất phát) và 1 đỉnh có bậc trong lớn hơn bậc ngoài 1 đơn vị (đỉnh kết thúc)

Chu trình Euler: là chu trình xuất phát từ 1 đỉnh, đi qua tất cả các cạnh của đồ thị, mỗi cạnh đi qua đúng một lần, và quay lại đỉnh xuất phát.

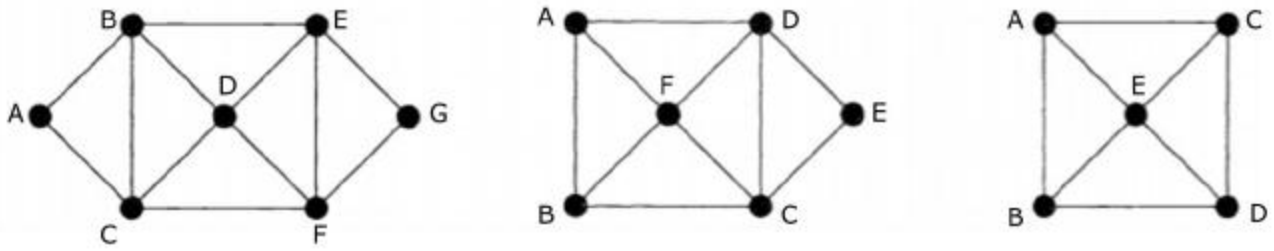
Điều kiện cần và đủ để tồn tại chu trình Euler

- Đồ thị vô hướng:
  - Đồ thị liên thông
  - Mọi đỉnh có bậc chẵn
- Đồ thị có hướng
  - Đồ thị liên thông
  - Mọi đỉnh có bậc ngoài bằng bậc trong

Đồ thị Euler: Đồ thị được gọi là đồ thị Euler nếu nó có chu trình Euler.

Đồ thị nửa Euler: Đồ thị được gọi là nửa Euler nếu nó có đường đi Euler. Mọi đồ thị Euler luôn là nửa Euler, nhưng điều ngược lại không luôn đúng.

Hình 7.1 mô tả ví dụ minh họa về đồ thị Euler và nửa Euler. Đồ thị trong hình 7.1.a là đồ thị Euler vì nó có chu trình Euler A, B, C, D, B, E, D, F, E, G, F, C, A. Đồ thị trong hình 7.1.b không có chu trình Euler nhưng có đường đi Euler A, D, E, C, D, F, A, B, F, C, B; vì vậy đồ thị đó là đồ thị nửa Euler. Đồ thị trong hình 7.1.c không có chu trình cũng như đường đi Euler.



(a) **(b)** **(c)**

Hình 7.1. (a) Đồ thị Euler, (b) Đồ thị nửa Euler,
(c) Đồ thị không chứa đường đi và chu trình Euler

Điều kiện cần và đủ để một đồ thị là đồ thị Euler được phát biểu thông qua các định lý và hệ quả sau:

Bổ đề 7.1: Đồ thị G có chu trình nếu bậc của mỗi đỉnh của đồ thị G không nhỏ hơn 2.

Định lý 7.2 (Euler 1736): Đồ thị vô hướng liên thông G là đồ thị Euler khi và chỉ khi mọi đỉnh của G đều có bậc chẵn.

Hệ quả 7.3: Đồ thị vô hướng liên thông G là đồ thị nửa Euler khi và chỉ khi đồ thị G có đúng hai đỉnh có bậc lẻ.

Định lý 7.4: Đồ thị có hướng liên thông mạnh G là đồ thị Euler khi và chỉ khi mọi đỉnh của G đều có nửa bậc trong bằng nửa bậc ngoài.

7.1.2 Giải thuật tìm chu trình/đường Euler

Để tìm chu trình/đường Euler của đồ thị G = (V, E) có thể sử dụng thuật toán Fleury sau:

1. Kiểm tra đồ thị G có không hoặc hai đỉnh bậc lẻ hay không. Nếu sai, thông báo đồ thị G không có đường Euler và kết thúc thuật toán. Ngược lại, sang bước 2.
2. Nếu đồ thị không có đỉnh lẻ, chọn một đỉnh bất kỳ của G làm đỉnh bắt đầu đường Euler. Nếu đồ thị có hai đỉnh bậc lẻ, chọn một trong hai đỉnh đó làm đỉnh bắt đầu đường Euler.
3. Xuất phát từ đỉnh đã chọn ở bước 2, đi theo các cạnh của G theo quy tắc sau:

a. Mỗi khi đi qua một cạnh, xóa bỏ cạnh vừa đi qua và ghi lại đỉnh theo thứ tự từ trái qua phải,

b. Chỉ đi qua cạnh bắc cầu (cạnh duy nhất nối hai thành phần liên thông của đồ thị) nếu không còn lựa chọn nào khác.

Giải thuật có thể được cài đặt dựa theo đoạn mã giả sau:

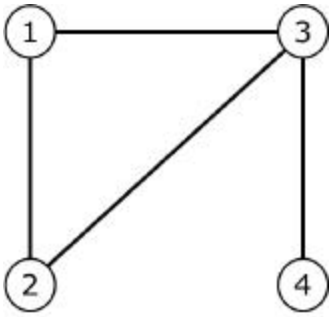
Dữ liệu vào: Đồ thị G = (V, E)

Kết quả: Tập danh sách các đỉnh nằm thuộc chu trình/đường Euler của đồ thị G.

```
Function Euler_Path
{
        Khởi tạo ngăn xếp S = ∅
        Khởi tạo danh sách chứa chu trình Euler cần tìm: C = ∅
        Chọn u là một đỉnh của đồ thị G
        Thêm u vào S
```

```
        while (S ≠ ∅) do
        {
            v = phần tử đầu của S
            if (v có đỉnh kề) then
            {
                w = đỉnh đầu tiên trong danh sách kề của u
                Thêm w vào S
                Loại bỏ cạnh (v, w) khỏi đồ thị G
            }
            else
            {
                Loại bỏ v khỏi S
                Thêm v vào CE
            }
        }
}
```

Ví dụ 7.1: Xác định đường Euler của đồ thị trong hình 7.2:



Hình 7.2: Ví dụ minh họa giải thuật Fleury

1. Kiểm tra đồ thị có thỏa mãn điều kiện để có chu trình Euler bằng cách tính số bậc của các đỉnh. Bậc của đỉnh 1 = 2, bậc của đỉnh 2 = 2, bậc của đỉnh 3 = 3, bậc của đỉnh 4 = 1. Do đồ thị có đúng 2 đỉnh có bậc lẻ, đồ thị có đường Euler.
2. Chọn một trong hai đỉnh có bậc lẻ làm đỉnh xuất phát của đường Euler cần tìm. Giả sử chọn đỉnh 3 để bắt đầu.

3. Chọn cạnh tiếp theo để đi theo nguyên tắc chỉ đi qua cạnh bắc cầu (cạnh duy nhất nối hai thành phần liên thông của đồ thị) nếu không còn lựa chọn nào khác, do vậy xuất phát từ đỉnh 3, lựa chọn một trong hai cạnh (3, 1) hoặc (3, 2) để thực hiện bước đi tiếp theo. Giả sử, lựa chọn cạnh (3, 1). Ghi nhận lại đỉnh 1 vào đường Euler → Đường Euler hiện giờ gồm các đỉnh {3, 1}. Xóa bỏ cạnh (3, 1) khỏi đồ thị.
4. Lặp lại bước 3 với đỉnh xuất phát là đỉnh 1. Do tại đỉnh 1 có duy nhất một cạnh (1, 2), do vậy lựa chọn cạnh này để thực hiện bước đi tiếp theo → Ghi nhận lại đỉnh 2 vào đường Euler → Đường Euler hiện giờ gồm các đỉnh {3, 1, 2}. Xóa bỏ cạnh (1, 2) khỏi đồ thị.
5. Lặp lại bước 3 với đỉnh xuất phát là đỉnh 2. Do tại đỉnh 2 có duy nhất một cạnh (2, 3), do vậy lựa chọn cạnh này để thực hiện bước đi tiếp theo → Ghi nhận lại đỉnh 3 vào đường Euler → Đường Euler hiện giờ gồm các đỉnh {3, 1, 2, 3}. Xóa bỏ cạnh (2, 3) khỏi đồ thị.
6. Lặp lại bước 3 với đỉnh xuất phát là đỉnh 3. Tại thời điểm này, tại đỉnh 3 có duy nhất một cạnh (3, 4), do vậy lựa chọn cạnh này để thực hiện bước đi tiếp theo → Ghi nhận lại đỉnh 4 vào đường Euler → Đường Euler hiện giờ gồm các đỉnh {3, 1, 2, 3, 4}. Xóa bỏ cạnh (3, 4) khỏi đồ thị.
7. Quá trình xây dựng đường Euler kết thúc do toàn bộ tập cạnh của đồ thị đã cho đã được xét. Đường Euler tìm được là {3, 1, 2, 3, 4}.

7.2 CHU TRÌNH HAMILTON

7.2.1 Định nghĩa

Cho đồ thị G = (V, E), có n đỉnh.

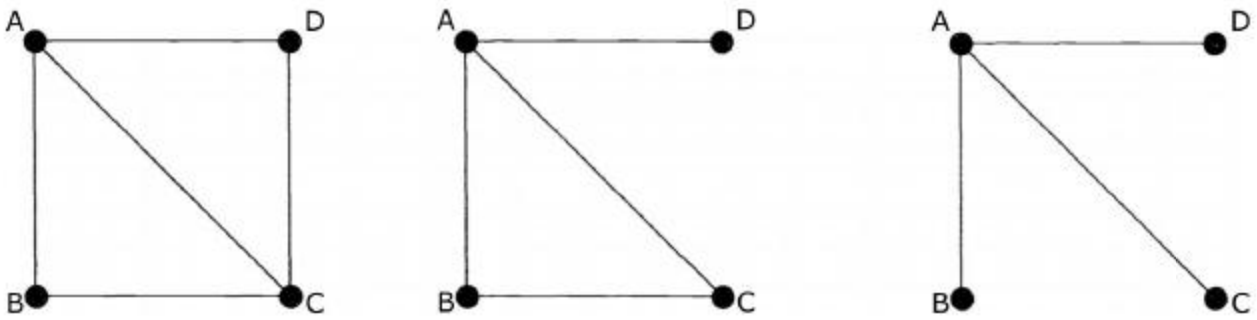
Đường đi Hamilton: là một đường đi đơn qua tất cả các đỉnh của đồ thị đúng 1 lần.

Chu trình Hamilton: chu trình đơn đi qua tất cả các đỉnh của đồ thị đúng 1 lần và quay về đỉnh xuất phát.

Đồ thị Hamilton: là đồ thị có chứa ít nhất 1 chu trình Hamilton.

Đồ thị nửa Hamilton: là đồ thị có chứa đường đi Hamilton. Mọi đồ thị Hamilton luôn là nửa Hamilton, nhưng điều ngược lại không luôn đúng.

Hình 7.3 mô tả ví dụ minh họa về đồ thị Hamilton và nửa Hamilton. Đồ thị trong hình 7.3.a là đồ thị Hamilton do nó có chu trình Hamilton A, B, C, D, A. Đồ thị trong hình 7.3.b không có chu trình Hamilton nhưng có đường Hamilton D, A, B, C, do đó đồ thị là nửa Hamilton. Đồ thị trong hình 7.3.c không có chu trình và đường Hamilton.



(a) **(b)** **(c)**

Hình 7.3. (a) Đồ thị Hamilton, (b) đồ thị nửa Hamilton, (c) đồ thị không chứa đường đi Hamilton

Khác với đồ thị Euler, không có điều kiện cần và đủ để một đồ thị là đồ thị Euler. Tuy nhiên, tồn tại điều kiện đủ để một đồ thị là đồ thị Hamilton như sau:

Định lý 7.5 (Ore, 1960): Nếu G là đơn đồ thị vô hướng có n đỉnh ($n \geq 3$) và với mọi cặp đỉnh u, v không kề nhau, tổng bậc của u và bậc của v lớn hơn hoặc bằng n thì G là đồ thị Hamilton.

Bổ đề 7.6 (Dirac, 1952): Nếu G là đơn đồ thị vô hướng có n đỉnh ($n \geq 3$) và mọi đỉnh v của G đều có bậc lớn hơn hoặc bằng n/2 thì G là đồ thị Hamilton.

Bổ đề 7.7 (Ghouila-Houiri, 1960): Nếu G là đồ thị có hướng liên thông mạnh có n đỉnh và mọi đỉnh của G có bậc lớn hơn hoặc bằng n thì G là đồ thị Hamilton.

Định lý 7.8 (Meyniel, 1973): Nếu G là đồ thị có hướng liên thông mạnh có n đỉnh và với mọi cặp đỉnh u, v không kề nhau, tổng bậc của u và bậc của v lớn hơn hoặc bằng (2n-1) thì G là đồ thị Hamilton.

7.2.2 Giải thuật tìm chu trình Hamilton

Để liệt kê tất cả các chu trình Hamilton của đồ thị G = (V, E), trong đó $V = \{v_0, v_1, \ldots, v_n\}$, sử dụng thuật toán quay lui với ý tưởng như sau:

1. Bắt đầu với đỉnh v_0 của đồ thị. Khởi tạo danh sách H = {v_0} dùng để chứa các đỉnh của chu trình Hamilton cần tìm.
2. Thêm các đỉnh khác vào H (bắt đầu từ đỉnh v_1) nếu đỉnh đó thỏa mãn điều kiện sau: đỉnh được thêm chưa có trong H, và kề với đỉnh vừa được thêm vào H ở bước ngay trước đó.

Giải thuật có thể được cài đặt dựa theo đoạn mã giả sau:

Dữ liệu vào: Đồ thị G = (V, E), và tập H chứa các đỉnh của chu trình Hamilton cần tìm: H = {v_0}, trong đó v_0 là một đỉnh bất kỳ của đồ thị.

Kết quả: tất cả các chu trình Hamilton của G.

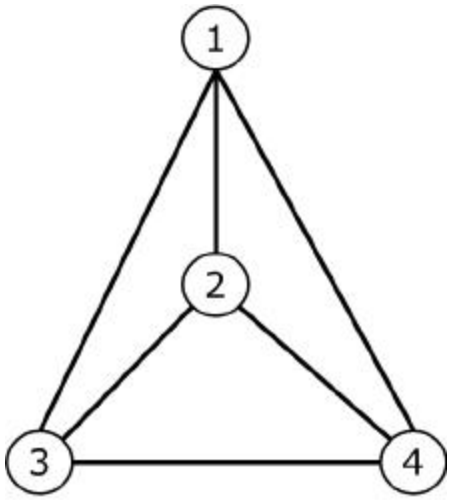
```
Function Hamilton(k)
{
        for đỉnh u kề với đỉnh H[k-1] do
        {
            if (k = n + 1 and u = v0) then write(H[1], …, H[n], v0)
            else
            {
                if (visited[u] == false) then
                {
                    H[k] = u
                    visited[u] = true
                    Hamilton(k + 1)
                    visited[u] = false
                }
            }
        }
}
```

Hàm Hamilton có thể được sử dụng như sau:

```
// main program
Function main
{
        for v ∈ V do visited[v] = false
        H[1] = v0   // v0 là một đỉnh bất kỳ của đồ thị
        visited[v0] = true
        Hamilton(2)
```

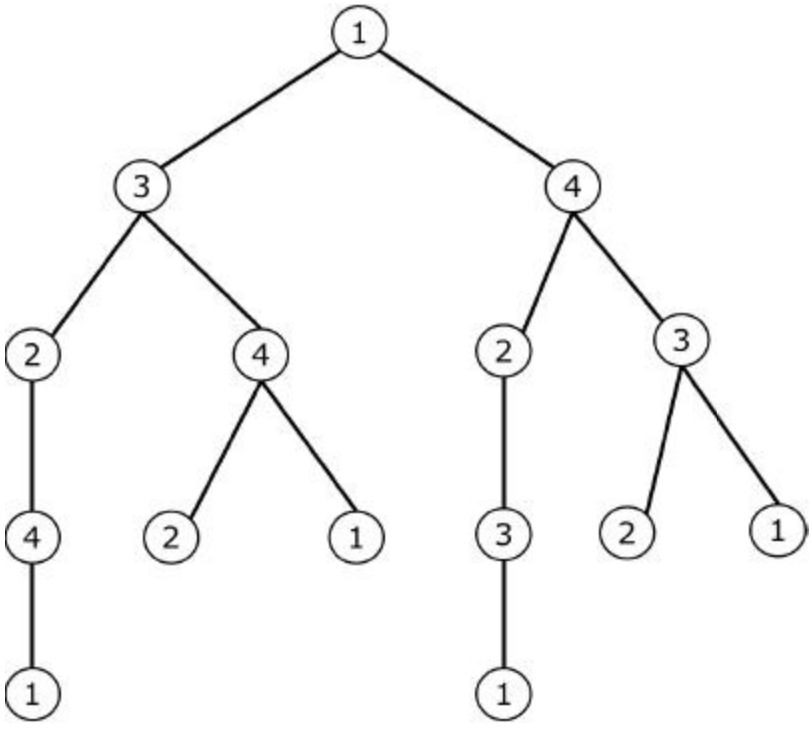
```
}
```

Ví dụ 7.2: Xác định chu trình Hamilton của đồ thị trong hình 7.4.



Hình 7.4: Ví dụ minh họa giải thuật xác định chu trình Hamilton

Áp dụng giải thuật mô tả ở trên ta xây dựng được cây liệt kê chu trình Hamilton như sau:

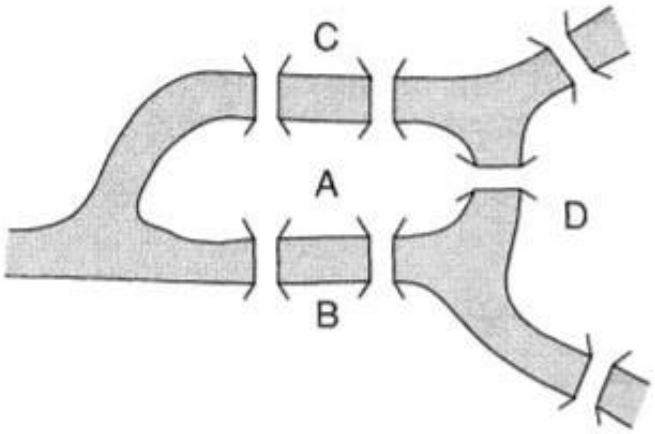


Hình 7.5: Cây liệt kê chu trình Hamilton tương ứng với đồ thị trong hình 7.4

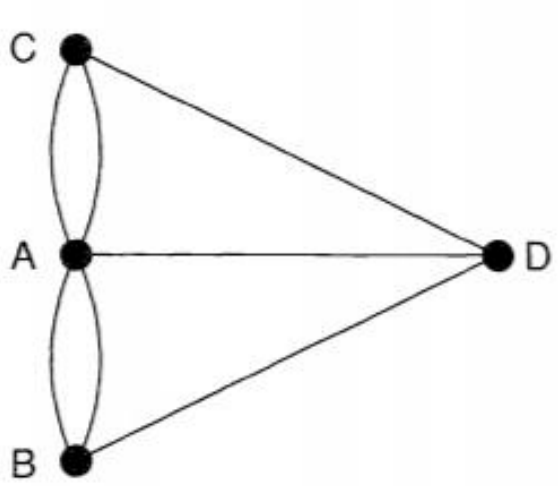
Dựa vào cây liệt kê chu trình Hamilton trong hình 7.5, ta xác định được hai chu trình Hamilton của đồ thị trong hình 7.4 như sau: {1, 3, 2, 4, 1} và {1, 4, 2, 3, 1}.

7.3 ỨNG DỤNG

Bài toán 1 (Bài toán bảy cầu ở Königsberg): thành phố Königsberg, Phổ (nay là Kaliningrad, Nga) bao gồm hai hòn đảo lớn nối với nhau và với đất liền bởi bảy cây cầu (Hình 7.6). Bài toán đặt ra là tìm một tuyến đường đi qua mỗi cây cầu đúng một lần và sau đó quay về điểm xuất phát. Năm 1736, nhà toán học Euler đã giải bài toán này bằng cách xây dựng đồ thị tương ứng với bản đồ trong hình 7.7. Sử dụng đồ thị trong hình 7.7 Euler đã chứng minh được bài toán trên là không giải được.

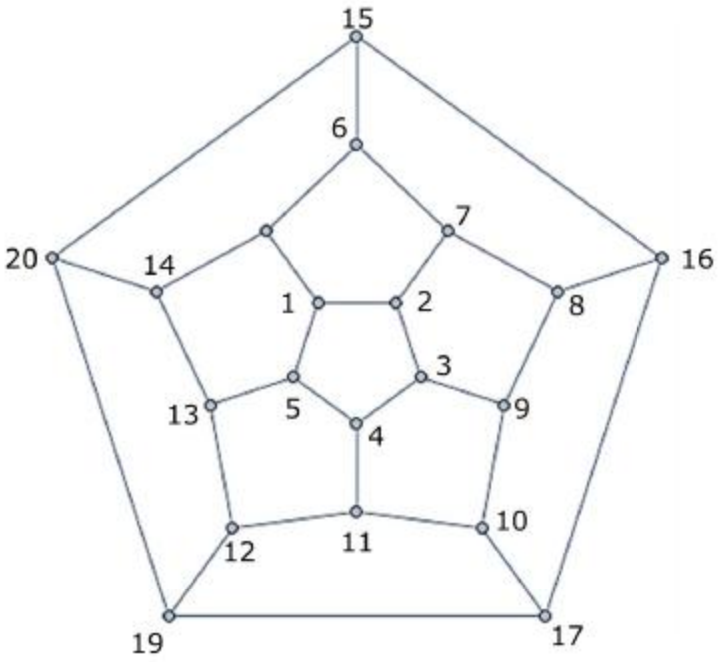


Hình 7.6: Bản đồ mô phỏng 7 cây cầu tại thành phố Königsberg



Hình 7.7: Đồ thị tương ứng với bản đồ trong hình 7.6

Bài toán 2 (Bài toán du lịch của Hamilton): Cho trước một danh sách gồm 20 thành phố và đường đi giữa các thành phố như trên hình 7.8. Bài toán đặt ra là có thể đi qua tất cả 20 thành phố, mỗi thành phố đúng một lần, rồi quay lại điểm xuất phát hay không? Và có tất cả bao nhiêu cách đi. Hamilton đã chỉ ra nguyên tắc di chuyển để giải bài toán này và lý thuyết về chu trình Hamilton sẽ giải quyết các bài toán tương tự như bài toán số 2.



Hình 7.8: Đồ thị mô tả đường đi giữa 20 thành phố

Bài toán 3 (Mã đi tuần): Đây là bài toán về việc di chuyển một quân mã trên bàn cờ vua (8 x 8). Quân mã được đặt ở một ô trên một bàn cờ trống nó phải di chuyển theo quy tắc của cờ vua để đi qua mỗi ô trên bàn cờ đúng một lần rồi trở về ô xuất phát. Bài toán này là một bài toán cụ thể của bài toán tìm chu trình Hamilton.

TÓM TẮT

Trong bài này, học viên cần nắm:

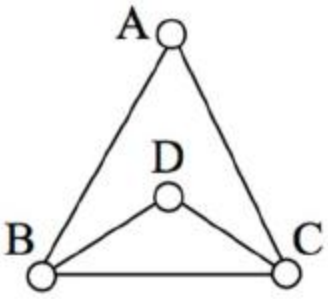
Khái niệm về chu trình Euler: là chu trình đơn đi qua mỗi cạnh của đồ thị đúng một lần, chu trình Hamilton: là chu trình đi qua tất cả các đỉnh của đồ thị, mỗi đỉnh đúng một lần.

Học viên cũng cần nắm được giải thuật tìm chu trình Euler và giải thuật tìm chu trình Hamilton, cũng như ứng dụng của việc tìm chu trình Euler và chu trình Hamilton trong thực tế.

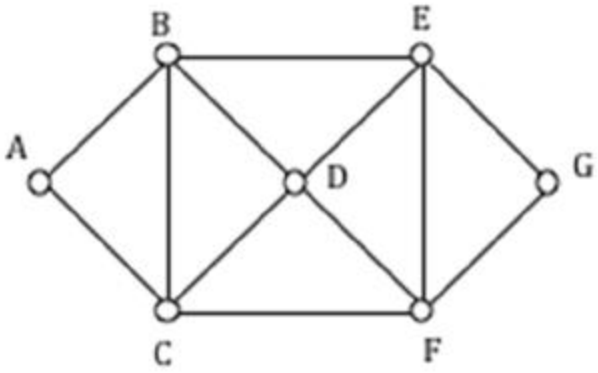
CÂU HỎI ÔN TẬP

Câu 1: Sử dụng thuật toán Fleury xác định chu trình Euler (nếu có) của các đồ thị sau:

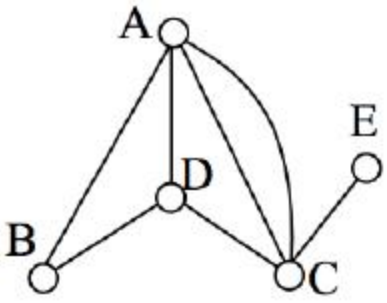
a.



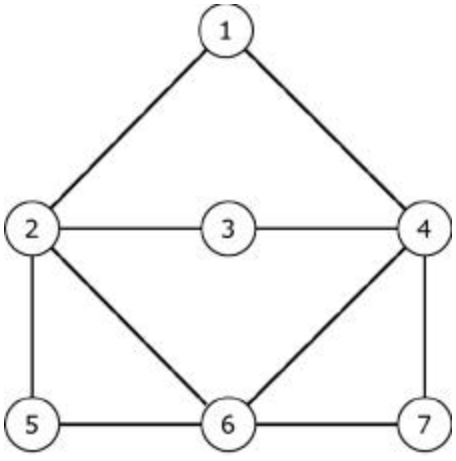
b.



c.



d.



Câu 2: Xác định chu trình Hamilton (nếu có) của các đồ thị trong câu 1.

Câu 3: Viết hàm kiểm tra một đồ thị là đồ thị Euler, nửa Euler hay không.

Câu 4: Viết hàm kiểm tra một đồ thị là đồ Hamilton, nửa Hamilton hay không.

Câu 5: Viết hàm tìm chu trình Euler của đồ thị.

Câu 6: Viết hàm tìm chu trình Hamilton của đồ thị.

BÀI 8: DUYỆT ĐỒ THỊ

Sau khi học xong bài này, học viên có thể:

- *Hiểu giải thuật duyệt đồ thị theo chiều rộng và theo chiều sâu.*
- *Vận dụng được các giải thuật duyệt đồ thị để giải các bài toán cụ thể.*

8.1 BÀI TOÁN DUYỆT ĐỒ THỊ

Trong lý thuyết đồ thị, bài toán duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó sao cho mỗi đỉnh chỉ được viếng thăm một lần là một bài toán quan trọng và có nhiều ứng dụng. Nhiều thuật toán trên đồ thị được xây dựng dựa trên cơ sở của bài toán duyệt đồ thị này. Để duyệt các đỉnh của đồ thị một cách có hệ thống, chúng ta cần phát triển các thuật toán duyệt đồ thị. Hai thuật toán duyệt đồ thị cơ bản được trình bày trong bài này gồm có thuật toán duyệt theo chiều rộng (Breadth First Search) và duyệt theo chiều sau (Depth First Search).

Giả sử G = (V, E) là đồ thị với tập đỉnh V và tập cạnh E, v_0 là một đỉnh bất kỳ của G. Thuật toán duyệt đồ thị tổng quát được phát biểu như sau:

Bước 1: Khởi tạo cấu trúc dữ liệu kiểu danh sách để chứa các đỉnh cần duyệt: L ← $\{v_0\}$.

Bước 2: Lấy đỉnh u ra khỏi đầu danh sách L.

Bước 3: Duyệt đỉnh u.

Bước 4: Thêm các đỉnh liền kề với u vào danh sách L.

Bước 5: Nếu L ≠ ∅, quay lại bước 2. Ngược lại, dừng thuật toán.

8.2 DUYỆT THEO CHIỀU RỘNG (BFS)

Trong giải thuật duyệt theo chiều rộng (Breath First Search), cấu trúc danh sách L mô tả trong giải thuật duyệt đồ thị tổng quát ở mục 8.1 được tổ chức theo kiểu hàng

đợi (queue). Với cách tổ chức danh sách L như vậy, việc duyệt cây có tính chất lan rộng. Nghĩa là đỉnh gần với đỉnh xuất phát sẽ được xét trước, đỉnh xa đỉnh xuất phát sẽ được xét sau.

Thuật toán duyệt đồ thị theo chiều rộng (BFS) được phát biểu như sau:

Dữ liệu vào: Đồ thị G = (V, E), đỉnh xuất phát v.

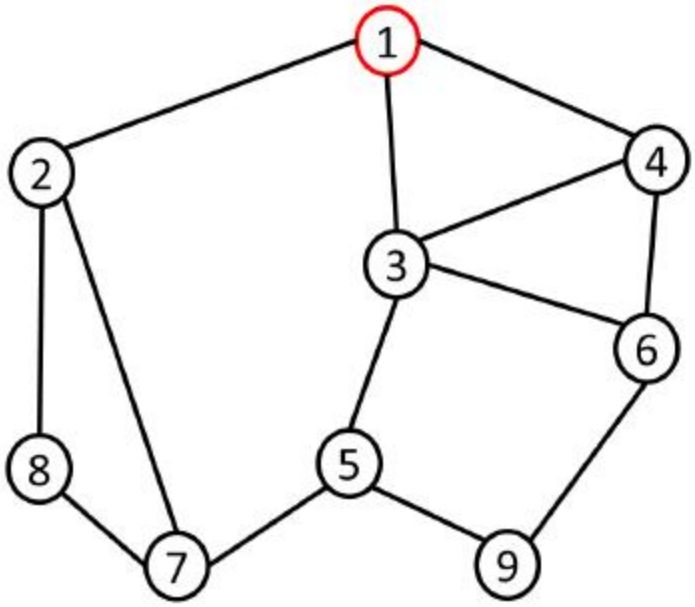
Kết quả: Danh sách các đỉnh của đồ thị G.

```
Function BFS(v)
{
        Khởi tạo hàng đợi L chỉ chứa đỉnh xuất phát v
        for u ∈ V do
                visited[u] = false
        visited[v] = true
        while (L ≠ ∅)
        {
              Loại u ra khỏi đầu hàng đợi L.
              Thăm đỉnh u.
              for trạng thái z kề u do
              {
                   if (visited[z] ≠ true)
                   {
                        Thêm z vào cuối hàng đợi L.
                        visited[z] = true
                   }
                   }
         }
}
```

Hàm BFS được gọi như sau:

```
Function main
{
          for v ∈ V do visited[v] = false
          for v ∈ V do
                if (visited[v] == false) then BFS(v)
}
```

Ví dụ **8.1**: Mô tả các bước duyệt đồ thị theo chiều rộng (BFS) với đồ thị vô hướng G = (V, E) trong hình 8.1. Đỉnh bắt đầu là đỉnh 1.



Hình 8.1: Ví dụ minh họa giải thuật duyệt đồ thị theo chiều rộng

Thuật toán duyệt đồ thị theo chiều rộng được mô tả trong bảng dưới đây:

Khởi tạo: Hàng đợi L = {1}, mảng visited = {1, 0, 0, 0, 0, 0, 0, 0, 0} (ký hiệu: 1 = true, 0 = false).

Đỉnh được duyệt (u)	Danh sách kề của u	Hàng đợi L	Mảng visited
1	{2, 3, 4}	{2, 3, 4}	{1, 1, 1, 1, 0, 0, 0, 0, 0}
2	{1, 7, 8}	{3, 4, 7, 8}	{1, 1, 1, 1, 0, 0, 1, 1, 0}
3	{1, 4, 5, 6}	{4, 7, 8, 5, 6}	{1, 1, 1, 1, 1, 1, 1, 1, 0}
4	{1, 3, 6}	{7, 8, 5, 6}	{1, 1, 1, 1, 1, 1, 1, 1, 0}
7	{2, 5, 8}	{8, 5, 6}	{1, 1, 1, 1, 1, 1, 1, 1, 0}
8	{2, 7}	{5, 6}	{1, 1, 1, 1, 1, 1, 1, 1, 0}
5	{3, 9}	{6, 9}	{1, 1, 1, 1, 1, 1, 1, 1, 1}
6	{3, 4, 9}	{9}	{1, 1, 1, 1, 1, 1, 1, 1, 1}
9	{5, 6}	∅	{1, 1, 1, 1, 1, 1, 1, 1, 1}

Thứ tự các đỉnh được duyệt của đồ thị trong hình 8.1 theo thuật toán duyệt theo chiều rộng là: 1, 2, 3, 4, 7, 8, 5, 6, 9.

8.3 DUYỆT THEO CHIỀU SÂU (DFS)

Trong giải thuật duyệt theo chiều sâu (Depth First Search), cấu trúc danh sách L mô tả trong giải thuật duyệt đồ thị tổng quát ở mục 8.1 được tổ chức theo kiểu ngăn xếp (stack). Với cách tổ chức danh sách L như vậy, mỗi lần duyệt một đỉnh, ta duyệt đến tận cùng của mỗi nhánh rồi mới duyệt sang nhánh tiếp theo.

Thuật toán duyệt đồ thị theo chiều sâu (DFS) được phát biểu như sau:

Dữ liệu vào: Đồ thị G = (V, E), đỉnh xuất phát v.

Kết quả: Danh sách các đỉnh của đồ thị G.

```
Function DFS(v)
{
      Khởi tạo ngăn xếp L chỉ chứa đỉnh xuất phát v
      visited[v] = true
       for u ∈ V do
                  visited[u] = false
      while (L ≠ ∅)
      {
            Loại u ra khỏi đầu hàng đợi L.
            Thăm đỉnh u.
            for trạng thái z kề u do
            {
                  if (visited[z] ≠ true)
                  {
                        Thêm z vào đầu ngăn xếp L.
                        visited[z] = true
                        }
            }
      }
}
```

Đồ thị được duyệt bằng hàm DFS theo cách sau:

```
Function main()

{
          for v ∈ V do visited[v] = false
          for v ∈ V do
                  if (visited[v] == false) then DFS(v)
}
```

Ví dụ 8.2: Mô tả các bước duyệt đồ thị theo chiều sâu (DFS) với đồ thị vô hướng G = (V, E) trong hình 8.1. Đỉnh bắt đầu là đỉnh 1.

Khởi tạo: Hàng đợi L = {1}, mảng visited = {1, 0, 0, 0, 0, 0, 0, 0, 0} (ký hiệu: 1 = true, 0 = false).

Đỉnh được duyệt (u)	Danh sách kề của u	Hàng đợi L	Mảng visited
1	{2, 3, 4}	{2, 3, 4}	{1, 1, 1, 1, 0, 0, 0, 0, 0}
2	{1, 7, 8}	{7, 8, 3, 4}	{1, 1, 1, 1, 0, 0, 1, 1, 0}
7	{2, 5, 8}	{5, 8, 3, 4}	{1, 1, 1, 1, 1, 0, 1, 1, 0}
5	{3, 7, 9}	{9, 8, 3, 4}	{1, 1, 1, 1, 1, 0, 1, 1, 1}
9	{5, 6}	{6, 8, 3, 4}	{1, 1, 1, 1, 1, 1, 1, 1, 1}
6	{3, 4, 9}	{8, 3, 4}	{1, 1, 1, 1, 1, 1, 1, 1, 1}
8	{2, 7}	{3, 4}	{1, 1, 1, 1, 1, 1, 1, 1, 1}
3	{1, 4, 5, 6}	{4}	{1, 1, 1, 1, 1, 1, 1, 1, 1}
4	{1, 3, 6}	∅	{1, 1, 1, 1, 1, 1, 1, 1, 1}

Thứ tự các đỉnh được duyệt của đồ thị trong hình 8.1 theo thuật toán duyệt theo chiều sâu là: 1, 2, 7, 5, 9, 6, 8, 3, 4.

8.4 ỨNG DỤNG

8.4.1 Bài toán tìm đường đi

Cho đồ thị G = (V, E), s và g là hai đỉnh của đồ thị G. Hãy tìm đường đi (nếu có) từ đỉnh s tới đỉnh g. Bài toán này được giải bằng cách cải tiến giải thuật duyệt cây theo chiều rộng hoặc chiều sâu như sau:

Dữ liệu vào: Đồ thị G = (V, E), đỉnh xuất phát s, đỉnh kết thúc g.

Kết quả: Đường đi từ s tới g.

```
Function BFS(s, g)
{
        Khởi tạo hàng đợi L chỉ chứa đỉnh xuất phát s
       for u ∈ V do
       {
```

```
                visited[u] = false
                parent[u] = -1
        }
         visited[s] = true
        while (L ≠ ∅)
        {
              Loại u ra khỏi đầu hàng đợi L.
             if (u == g) then
                   {Thông báo tìm kiếm thành công. Dừng}
              for trạng thái z kề u do
              {
                   if (visited[z] ≠ true)
                   {
                        Thêm z vào cuối hàng đợi L.
                        visited[z] = true
                        parent[z] = u
                   }
               }
    }
     if L = ∅ then {thông báo tìm kiếm thất bại}
}
Function DFS(s, g)
{
      Khởi tạo ngăn xếp L chỉ chứa đỉnh xuất phát v
       for u ∈ V do
       {
                visited[u] = false
                parent[u] = -1
       }
        visited[s] = true
      while (L ≠ ∅)
      {
            Loại u ra khỏi đầu hàng đợi L.
            if (u == g) then
                  {Thông báo tìm kiếm thành công. Dừng}
            for trạng thái z kề u do
            {
```

```
                if (visited[z] ≠ true)
                {
                    Thêm z vào đầu ngăn xếp L.
                    visited[z] = true
                    parent[z] = u
                }
            }
        }
    }
```

Đường đi (nếu có) sẽ được khôi phục như sau: g ← v1 = parent[g] ← v2 = parent[v1] ← v3 = parent[v2] ← ... ← s.

Ví dụ 8.3: Mô tả các bước tìm đường đi từ đỉnh 1 đến đỉnh 7 theo thuật toán duyệt đồ thị theo chiều rộng (BFS) với đồ thị vô hướng G = (V, E) trong hình 8.1.

Khởi tạo: Hàng đợi L = {1}, mảng visited = {1, 0, 0, 0, 0, 0, 0, 0, 0} (ký hiệu: 1 = true, 0 = false), mảng parent = {-1, -1, -1, -1, -1, -1, -1, -1, -1}.

Đỉnh được duyệt (u)	**Danh sách** kề của u	Hàng đợi L	Mảng visited	Mảng parent
1	{2, 3, 4}	{2, 3, 4}	{1, 1, 1, 1, 0, 0, 0, 0, 0}	{-1, 1, 1, 1, 0, 0, 0, 0, 0}
2	{1, 7, 8}	{3, 4, 7, 8}	{1, 1, 1, 1, 0, 0, 1, 1, 0}	{-1, 1, 1, 1, 0, 0, 2, 2, 0}
3	{1, 4, 5, 6}	{4, 7, 8, 5, 6}	{1, 1, 1, 1, 1, 1, 1, 1, 0}	{-1, 1, 1, 1, 3, 3, 2, 2, 0}
4	{1, 3, 6}	{7, 8, 5, 6}	{1, 1, 1, 1, 1, 1, 1, 1, 0}	{-1, 1, 1, 1, 3, 3, 2, 2, 0}
7	{2, 5, 8}	{8, 5, 6}	{1, 1, 1, 1, 1, 1, 1, 1, 0}	{-1, 1, 1, 1, 3, 3, 2, 2, 0}

Thuật toán dừng do gặp trạng thái kết thúc. Đường đi từ đỉnh 1 đến đỉnh 7 được xác định bằng cách dò quay lui từ đỉnh 7 trong mảng parent. Đường đi tìm được như sau: 1, 2, 7.

Ví dụ 8.4: Mô tả các bước tìm đường đi từ đỉnh 1 đến đỉnh 7 theo thuật toán duyệt đồ thị theo chiều sâu (DFS) với đồ thị vô hướng G = (V, E) trong hình 8.1.

Khởi tạo: Hàng đợi L = {1}, mảng visited = {1, 0, 0, 0, 0, 0, 0, 0, 0} (ký hiệu: 1 = true, 0 = false), mảng parent = {-1, -1, -1, -1, -1, -1, -1, -1, -1}.

Đỉnh được duyệt (u)	Danh sách kề của u	Hàng đợi L	Mảng visited	Mảng parent
1	{2, 3, 4}	{2, 3, 4}	{1, 1, 1, 1, 0, 0, 0, 0, 0}	{-1, 1, 1, 1, 0, 0, 0, 0, 0}
2	{1, 7, 8}	{7, 8, 3, 4}	{1, 1, 1, 1, 0, 0, 1, 1, 0}	{-1, 1, 1, 1, 0, 0, 2, 2, 0}
7	{2, 5, 8}	{5, 8, 3, 4}	{1, 1, 1, 1, 1, 0, 1, 1, 0}	{-1, 1, 1, 1, 7, 0, 2, 2, 0}

Thuật toán dừng do gặp trạng thái kết thúc. Đường đi từ đỉnh 1 đến đỉnh 7 được xác định bằng cách dò quay lui từ đỉnh 7 trong mảng parent. Đường đi tìm được như sau: 1, 2, 7.

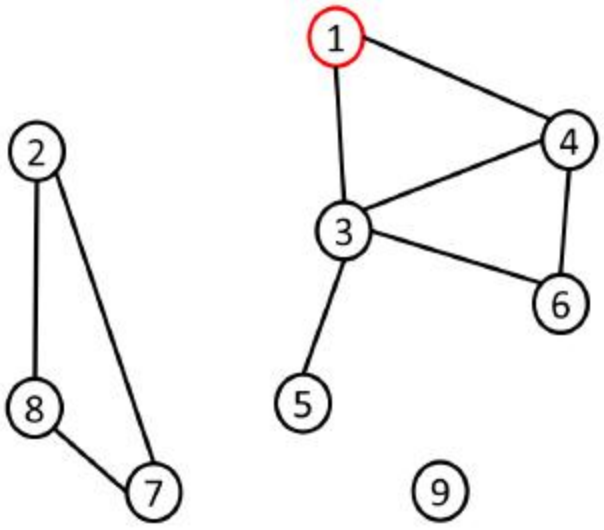
8.4.2 Bài toán tìm các thành phần liên thông

Cho đồ thị G = (V, E), hãy tìm các thành phần liên thông của đồ thị G. Tương tự bài toán tìm đường đi giữa hai đỉnh của đồ thị, bài toán tìm các thành phần liên thông cũng được giải bằng cách cải tiến giải thuật duyệt cây theo chiều rộng hoặc chiều sâu như sau:

```
Function BFS(v, inconnect)
{
        Khởi tạo hàng đợi L chỉ chứa đỉnh xuất phát v
        visited[v] = inconnect              // xác định thành phần liên thông của đỉnh v
        while (L ≠ ∅)
        {
                Loại u ra khỏi đầu hàng đợi L.
                for trạng thái z kề u do
                {
                        if (visited[z] == 0)
                        {
                                Thêm z vào cuối hàng đợi L.
```

```
                        visited[z] = inconnect
                        //parent[z] = u
                }
            }
        }
}

Function Find_connected
{
    for v ∈ V do visited[v] = 0
    count = 0       // biến dùng để đếm số lượng thành phần liên thông của G
    for v ∈ V do
        {
            if (visited[v] == 0) then
            {
                count = count + 1
                BFS(v, count)
            }
        }
}
```

Ví dụ 8.5: Mô tả các bước của giải thuật xác định các thành phần liên thông của đồ thị vô hướng G = (V, E) trong hình 8.2.



Hình 8.2: Ví dụ minh họa giải thuật xác định các thành phần liên thông

Khởi tạo: Mảng visited = {0, 0, 0, 0, 0, 0, 0, 0, 0}, inconnect = 0.

1. Xét v = 1 → count = 1, các bước thuật toán BFS(v, count) như sau: L = {1}, visited[1] = 1.

Đỉnh được duyệt (u)	Danh sách kề của u	Hàng đợi L	Mảng visited
1	{3, 4}	{3, 4}	{1, 0, 1, 1, 0, 0, 0, 0, 0}
3	{1, 4, 5, 6}	{4, 5, 6}	{1, 0, 1, 1, 1, 1, 0, 0, 0}
4	{1, 3, 6}	{5, 6}	{1, 0, 1, 1, 1, 1, 0, 0, 0}
5	{3}	{6}	{1, 0, 1, 1, 1, 1, 0, 0, 0}
6	{3, 4}	∅	{1, 0, 1, 1, 1, 1, 0, 0, 0}

Bước 1 dừng do danh sách L = ∅. Tìm được thành phần liên thông đầu tiên gồm các đỉnh {1, 3, 4, 5, 6}.

2. Xét v = 2 → count = 2, các bước thuật toán BFS(v, count) như sau: L = {2}, visited[2] = 2.

Đỉnh được duyệt (u)	Danh sách kề của u	Hàng đợi L	Mảng visited
2	{7, 8}	{7, 8}	{1, 2, 1, 1, 1, 1, 2, 2, 0}
7	{2, 8}	{8}	{1, 2, 1, 1, 1, 1, 2, 2, 0}
8	{2, 7}	∅	{1, 2, 1, 1, 1, 1, 2, 2, 0}

Bước 2 dừng do danh sách L = ∅. Tìm được thành phần liên thông thứ hai gồm các đỉnh {2, 7, 8}.

3. Xét v = 9 → count = 3, các bước thuật toán BFS(v, count) như sau: L = {9}, visited[9] = 3.

Đỉnh được duyệt (u)	Danh sách kề của u	Hàng đợi L	Mảng visited
9	∅	∅	{1, 2, 1, 1, 1, 1, 2, 2, 3}

Bước 3 dừng do danh sách L = ∅. Tìm được thành phần liên thông thứ ba gồm đỉnh {9}.

TÓM TẮT

Trong bài này, học viên cần nắm:

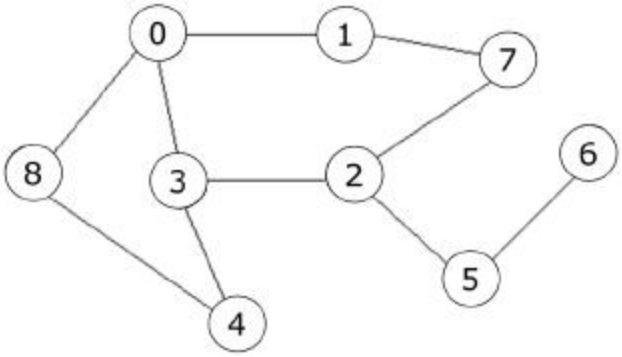
Bài toán duyệt đồ thị là bài duyệt tất cả các đỉnh có thể đến được từ một đỉnh xuất phát nào đó sao cho mỗi đỉnh chỉ được viếng thăm một lần. Đây là bài toán quan trọng và có nhiều ứng dụng của lý thuyết đồ thị. Hai thuật toán cơ bản để duyệt đồ thị là thuật toán duyệt đồ thị theo chiều rộng (breath first search) và theo chiều sâu (depth first search).

Học viên cũng cần nắm được ứng dụng của hai giải thuật duyệt đồ thị trong bài toán tìm đường đi và tìm các thành phần liên thông của đồ thị.

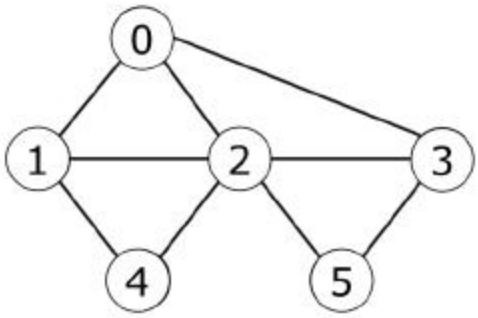
CÂU HỎI ÔN TẬP

Câu 1: Áp dụng giải thuật duyệt cây theo chiều rộng (BFS) với các đồ thị sau. Đỉnh xuất phát là đỉnh 0.

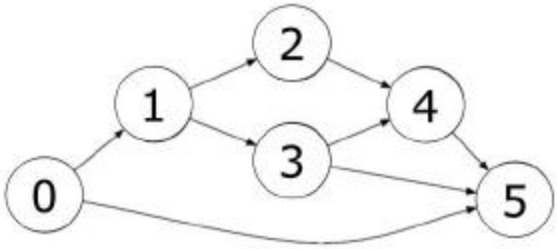
a.



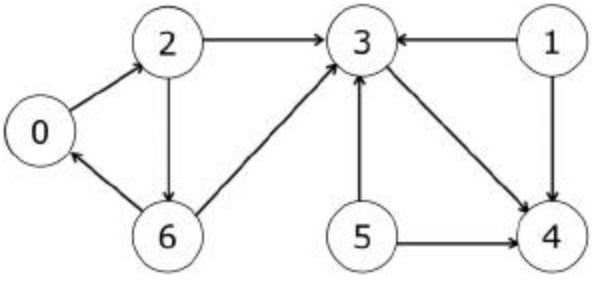
b.

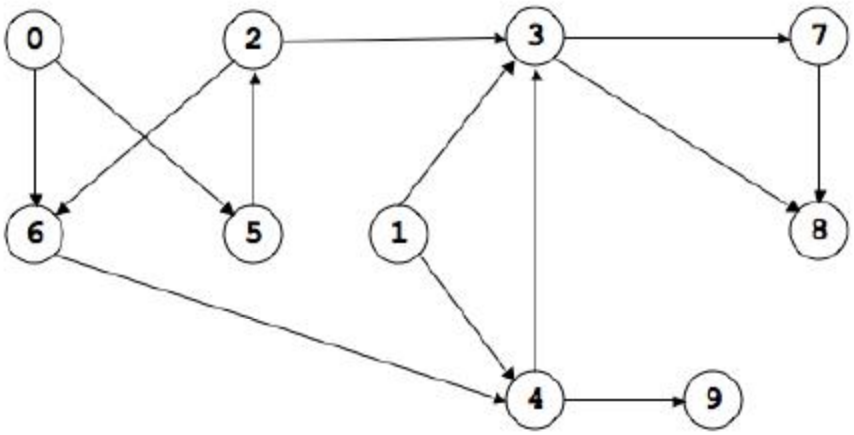


c.



d.

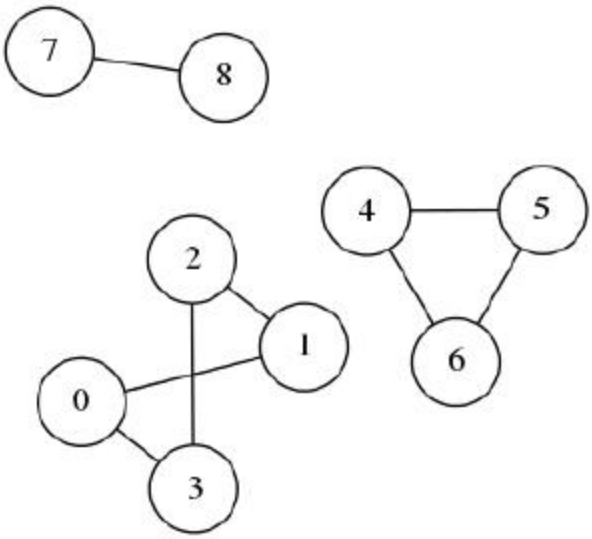




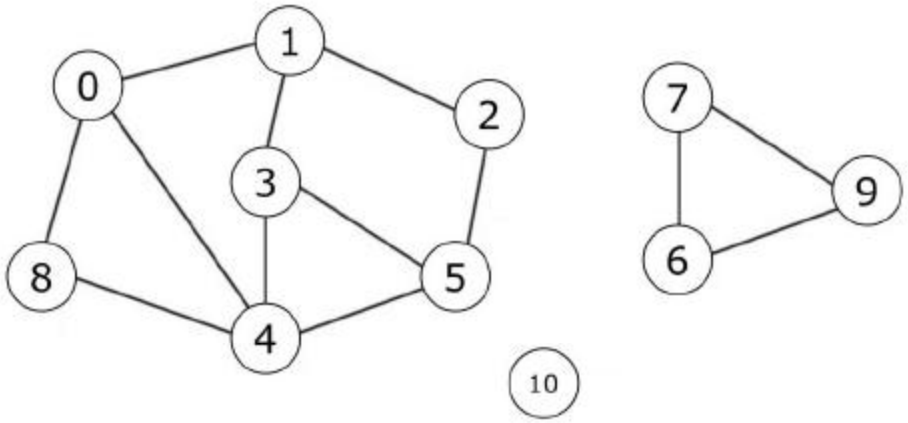
e.

Câu 2: Áp dụng giải thuật duyệt cây theo chiều sâu (DFS) với các đồ thị ở câu 1. Đỉnh xuất phát là đỉnh 0.

Câu 3: Áp dụng giải thuật tìm các thành phần liên thông của đồ thị sau:



a.



b.

Câu 4: Viết mã giả của giải thuật xác định các thành phần liên thông dựa theo giải thuật tìm kiếm theo chiều sâu.

Câu 5: Viết hàm duyệt đồ thị theo chiều rộng.

Câu 6: Viết hàm duyệt đồ thị theo chiều sâu.

Câu 7: Viết hàm tìm đường đi giữa 2 đỉnh của đồ thị.

Câu 8: Viết hàm xác định các thành phần liên thông của đồ thị.

BÀI 9: CÂY BAO TRÙM VÀ CÂY BAO TRÙM NHỎ NHẤT

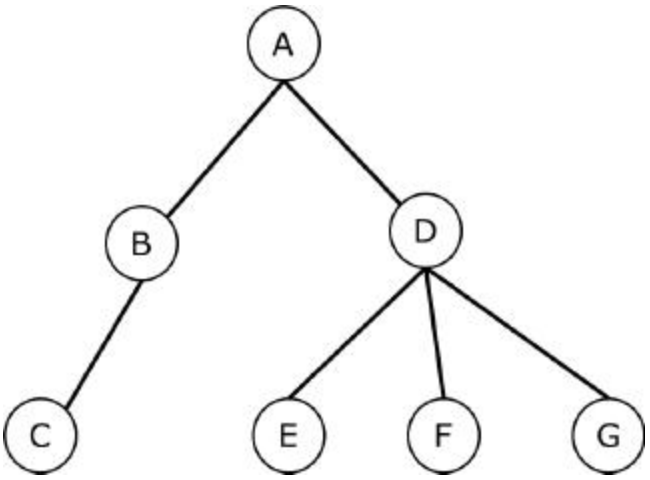
Sau khi học xong bài này, học viên có thể:

- *Hiểu được khái niệm cây, cây bao trùm, và cây bao trùm nhỏ nhất của đồ thị.*
- *Hiểu và lập trình được các giải thuật xác định cây bao trùm nhỏ nhất, bao gồm: giải thuật Kruskal và giải thuật Prim.*
- *Vận dụng được các giải thuật xác đinh cây bao trùm nhỏ nhất để giải các bài toán cụ thể.*

9.1 ĐỊNH NGHĨA

9.1.1 Cây và bụi

Cho đồ thị vô hướng $T = (V_T, E_T)$ với V_T là tập đỉnh gồm n đỉnh, E_T là tập cạnh. Đồ thị T được gọi là một cây nếu nó liên thông và không có chu trình. Ví dụ về cây được minh họa trong hình 9.1.

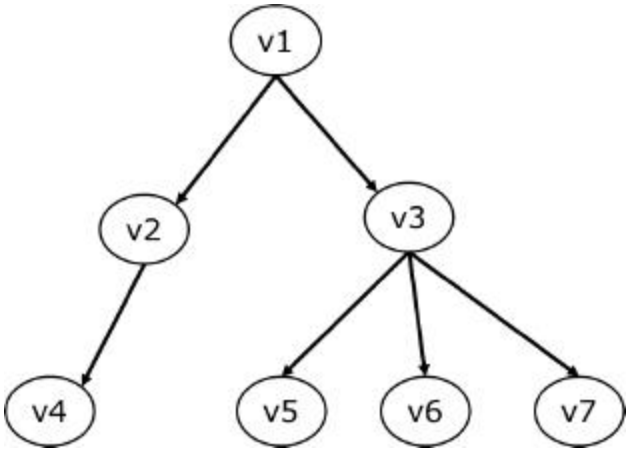


Hình 9.1: Ví dụ về cây

Đồ thị hữu hạn có hướng $G = (V_G, E_G)$ là một bụi gốc $v1 \in V_G$ nếu nó có ít nhất hai đỉnh và thỏa mãn các điều kiện sau:

1. Mỗi đỉnh khác v1 là điểm cuối của một cung duy nhất.
2. Đỉnh v1 không là điểm cuối của bất kỳ cung nào.
3. Đồ thị G không có chu trình.

Ví dụ về bụi được minh họa trong hình 9.2.



Hình 9.2: Ví dụ về bụi

Định lý 9.1: Giả sử T là đồ thị có n đỉnh. Khi đó, các phát biểu sau là tương đương:

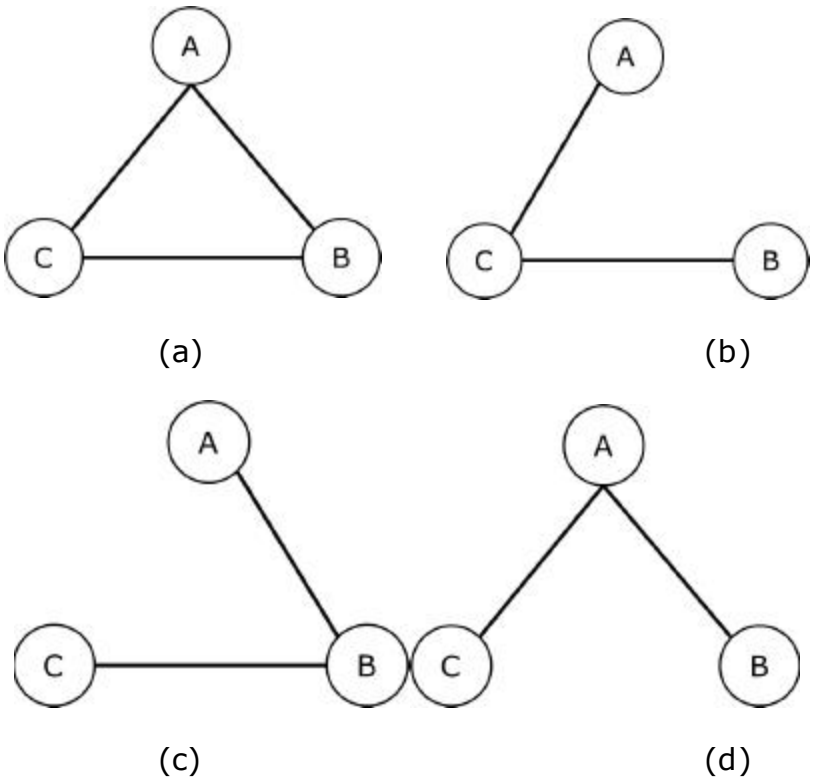
1. T là cây.
2. Cây T không có chu trình và có n – 1 cạnh.
3. Cây T liên thông và có n – 1 cạnh.
4. Cây T liên thông và nếu xóa bớt đi một cạnh bất kỳ thì T trở thành đồ thị không liên thông.
5. Hai đỉnh bất kỳ của T được nối với nhau bằng đúng một đường đơn.
6. Cây T không chứa chu trình, nhưng nếu thêm vào cây T một cạnh bất kỳ thì sẽ xuất hiện chu trình.

Định lý 9.2: Một cây có ít nhất hai đỉnh treo.

Định lý 9.3: Khi bỏ đi hướng của các cạnh, mọi bụi đều trở thành cây.

9.1.2 Cây bao trùm của đồ thị

Khái niệm cây bao trùm: Cho đồ thị vô hướng G = (V, E) với n đỉnh (n ≥ 2), đồ thị T được gọi là cây bao trùm (cây khung) của G nếu T là đồ thị bộ phận của G và T liên thông. Hình 9.3 là ví dụ về cây bao trùm của đồ thị.



Hình 9.3: (a) Đồ thị vô hướng G gồm 3 đỉnh và 3 cạnh, (b), (c), (d) Cây bao trùm của G.

Một số tính chất:

1. Đồ thị vô hướng G có cây bao trùm khi và chỉ khi G liên thông.
2. Số cây bao trùm (cây khung) của đồ thị vô hướng đầy đủ n đỉnh là n^{n-2}.
3. Đồ thị vô hướng liên thông G có ít nhất một cây bao trùm.
4. Mọi cây bao trùm của đồ thị G có cùng số lượng đỉnh và cạnh.
5. Giả sử đồ thị G có n đỉnh, cây bao trùm của G có n-1 cạnh.

Một số ứng dụng của cây bao trùm: Cây bao trùm thường được sử dụng để tìm đường đi ngắn nhất giữa các đỉnh của đồ thị. Một số ứng dụng của cây bao trùm gồm có: bài toán quy hoạch hạ tầng thành phố, bài toán định tuyến, ...

Thuật toán tìm cây bao trùm: Cây bao trùm của đồ thị vô hướng G có thể được xác định dựa theo thuật toán duyệt đồ thị theo chiều rộng hoặc chiều sâu. Hai thuật toán lần lượt được trình bày dưới đây.

Dữ liệu vào: Đồ thị vô hướng liên thông G = (V, E), một đỉnh bất kỳ r ∈ V, tập T = ∅ chứa tập cạnh của cây bao trùm cần tìm.

Kết quả: tập cạnh T của cây bao trùm.

Thuật toán xác định cây bao trùm theo chiều sâu:

```
Function SpanningTree_DFS(r):
{
        visited[r] = true
        For mỗi trạng thái u kề r do
        {
            If (visited[u] == false) then
            {
                T = T ∪ (r, u)
                SpanningTree_DFS(u)
            }
        }
}
```

Thuật toán xác định cây bao trùm theo chiều rộng:

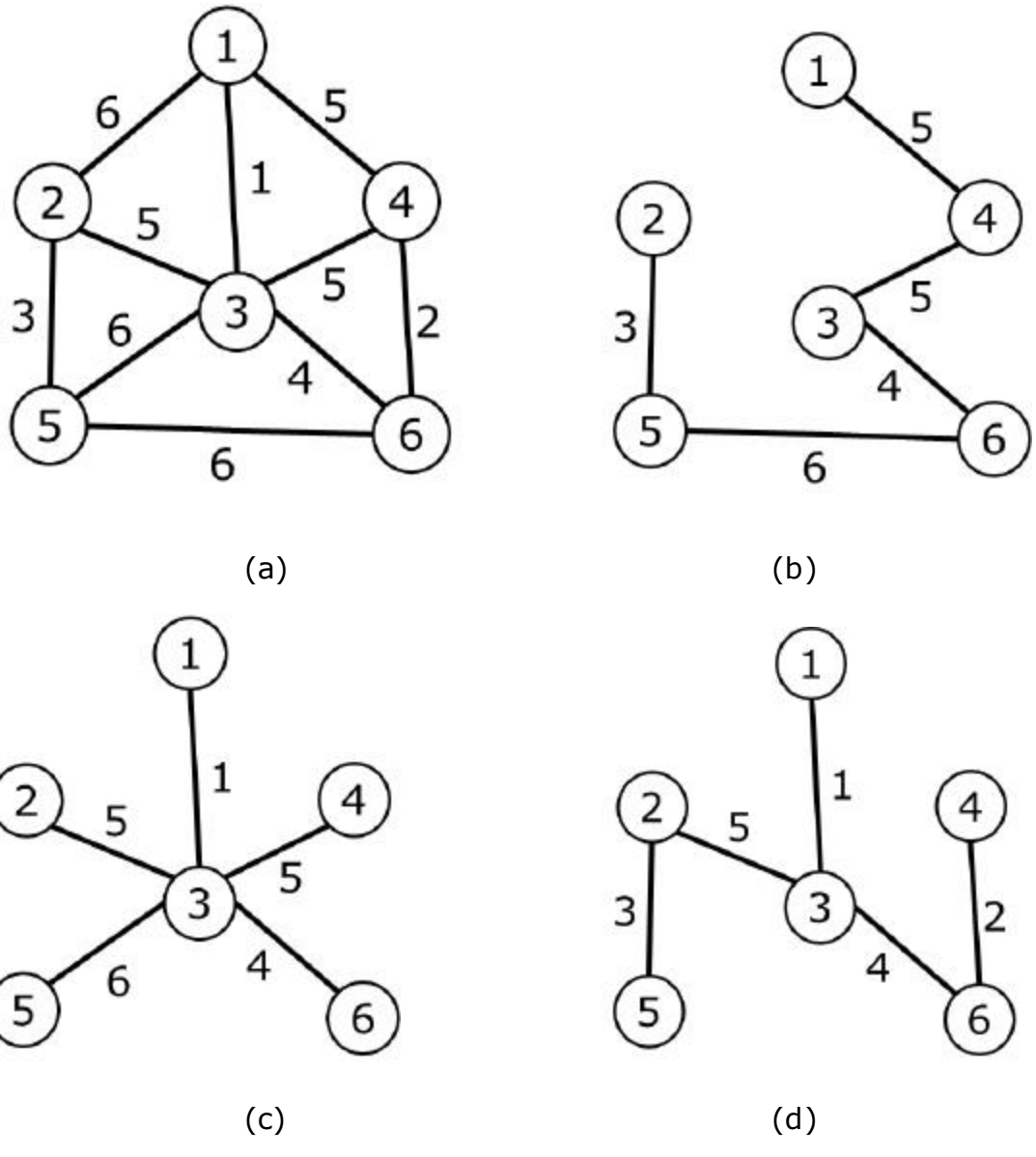
```
Function SpanningTree_BFS(r):
{
        Khởi tạo hàng đợi Q = ∅
        Thêm đỉnh r vào cuối hàng đợi Q.
        visited[r] = true
        while Q ≠ ∅ do
        {
            Loại trạng thái u ở đầu danh sách Q.
            for mỗi trạng thái v kề u do
            {
                if (visited[v] == false)
                {
                    Thêm đỉnh v vào cuối hàng đợi Q
                    visited[v] = false
                    T = T ∪ (u, v)
                }
            }
        }
}
```

9.1.3 Cây bao trùm nhỏ nhất

Cho đồ thị vô hướng liên thông có trọng số G = (V_G, E_G) với tập đỉnh V_G = {1, 2, …, n} và tập cạnh E_G gồm m cạnh.

Mỗi cạnh e bất kỳ của G có trọng số là c(e). Giả sử T = (V_T, E_T) là cây bao trùm của đồ thị G. Đại lượng $c(T) = \sum_{e \in E_T} c(e)$ được gọi là trọng số của cây bao trùm T. Cây bao trùm nhỏ nhất của G là cây bao trùm có trọng số bé nhất.

Hình 9.4 là ví dụ về cây bao trùm và cây bao trùm nhỏ nhất của đồ thị có trọng số.



Hình 9.4. (a) Đồ thị vô hướng có trọng số G, (b) Cây bao trùm của G, trọng số của cây là 23, (c) Cây bao trùm của G, trọng số của cây là 21, (d) Cây bao trùm nhỏ nhất của G, trọng số của cây là 15

Để tìm cây bao trùm nhỏ nhất của một đồ thị có n đỉnh, chúng ta có thể liệt kê tất cả các cây bao trùm của đồ thị đó và chọn ra cây bao trùm có trọng số nhỏ nhất. Tuy nhiên, giải thuật đó có độ phức tạp n^{n-2} trong trường hợp đồ thị là đầy đủ. Do vậy, giải thuật không khả thi cho dù đồ thị G có số đỉnh nhỏ. Hai giải thuật hiệu quả để xác định cây bao trùm nhỏ nhất là giải thuật Kruskal và Prim sẽ được trình bày dưới đây.

9.2 THUẬT TOÁN KRUSKAL

9.2.1 Mô tả

Thuật toán Kruskal xác định cây bao trùm nhỏ nhất của đồ thị G dựa trên chiến lược tham lam. Thuật toán gồm các bước cơ bản như sau:

1. Loại bỏ khuyên và các cạnh song song của đồ thị. Đối với các cạnh song song, chỉ giữ lại cạnh có trọng số nhỏ nhất.
2. Sắp xếp các cạnh theo thứ tự tăng dần của trọng số.
3. Khởi tạo tập cạnh T của cây bao trùm nhỏ nhất cần tìm bằng ∅.
4. Chọn cạnh có trọng số nhỏ nhất. Thêm cạnh vừa chọn vào tập cạnh T nếu cạnh vừa chọn không tạo thành chu trình với tập cạnh T đã có.
5. Lặp lại bước 4 cho tới khi tập cạnh T có n-1 cạnh (trong đó, n là số đỉnh của đồ thị đã cho).

Mã giả của giải thuật:

Dữ liệu vào: Đồ thị vô hướng liên thông $G = (V_G, E_G)$ với tập đỉnh V_G gồm n đỉnh và tập cạnh E_G gồm m cạnh.

Kết quả: Tập cạnh E_T của cây bao trùm nhỏ nhất của G.

```
Function Kruskal
{
        E_T = ∅
        while (|E_T | < n-1) and (E_G ≠ ∅) do
        {
            Chọn e là cạnh có trọng số nhỏ nhất trong E_G.
            if (E_T ∪ {e} không chứa chu trình) then E_T = E_T ∪ {e}
```

Loại cạnh e vừa chọn khỏi tập E_G.

}

if ($|E_T| < n-1$) then {Thông báo không tìm được cây bao trùm nhỏ nhất.}

}

Ví dụ 9.1: Tìm cây bao trùm nhỏ nhất T của đồ thị $G = (V_G, E_G)$ trong hình 9.4.a bằng thuật toán Kruskal.

Khởi tạo: tập cạnh của T là $E_T = \emptyset$

1. Chọn cạnh có trọng số nhỏ nhất trong E_G: e = (1, 3)

   Do ($E_T \cup e$) không chứa chu trình, E_T = {(1, 3)}

   Loại cạnh (1, 3) khỏi tập cạnh E_G của G.

2. Chọn cạnh có trọng số nhỏ nhất trong E_G: e = (4, 6)

   Do ($E_T \cup e$) không chứa chu trình, E_T = {(1, 3), (4, 6)}

   Loại cạnh (4, 6) khỏi tập cạnh E_G của G.

3. Chọn cạnh có trọng số nhỏ nhất trong E_G: e = (2, 5)

   Do ($E_T \cup e$) không chứa chu trình, E_T = {(1, 3), (4, 6), (2, 5)}

   Loại cạnh (2, 5) khỏi tập cạnh E_G của G.

4. Chọn cạnh có trọng số nhỏ nhất trong E_G: e = (3, 6)

   Do ($E_T \cup e$) không chứa chu trình, E_T = {(1, 3), (4, 6), (2, 5), (3, 6)}

   Loại cạnh (3, 6) khỏi tập cạnh E_G của G.

5. Chọn cạnh có trọng số nhỏ nhất trong E_G: e = (2, 3)

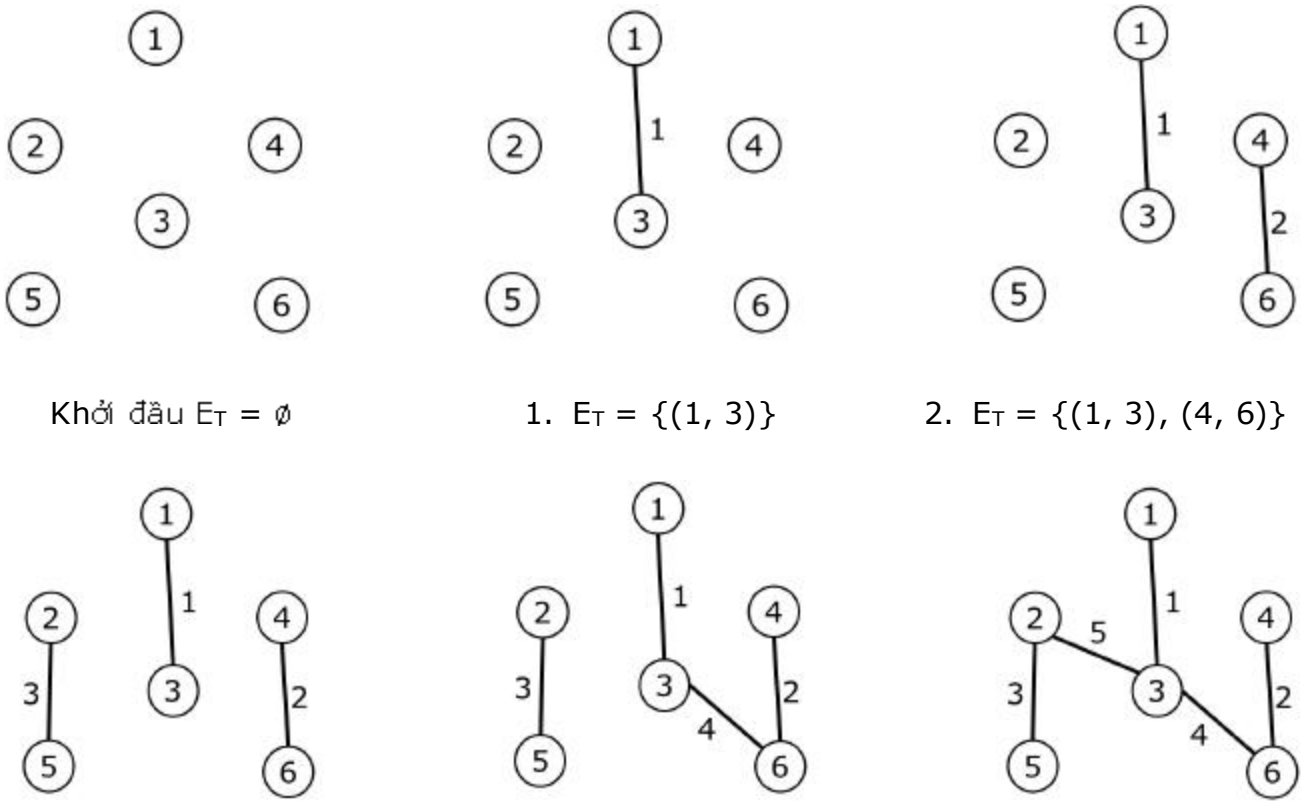
   Do ($E_T \cup e$) không chứa chu trình, E_T = {(1, 3), (4, 6), (2, 5), (3, 6), (2, 3)}

   Loại cạnh (3, 6) khỏi tập cạnh E_G của G.

6. Thuật toán dừng do số cạnh trong ET = n – 1 (với n là số đỉnh của đồ thị G).

Cây bao trùm nhỏ nhất tìm được có tập cạnh là E_T = {(1, 3), (4, 6), (2, 5), (3, 6), (2, 3)}.

Các bước của thuật toán có thể được minh họa bằng hình ảnh như sau:



Hình 9.5: Minh họa giải thuật Kruskal

9.3 THUẬT TOÁN PRIM

9.3.1 Mô tả

Tương tự giải thuật Kruskal, thuật toán Prim cũng dựa trên chiến lược tham lam. Xuất phát với một cây bao trùm rỗng, thuật toán Prim duy trì hai tập đỉnh, trong đó tập thứ nhất chứa các đỉnh đã được thêm vào cây bao trùm nhỏ nhất, và tập thứ hai chứa các đỉnh chưa được thêm vào cây bao trùm nhỏ nhất. Thuật toán lần lượt xem xét các cạnh nối hai tập đỉnh nói trên, và chọn cạnh có trọng số nhỏ nhất. Đỉnh cuối của cạnh vừa chọn sẽ được thêm vào tập đỉnh của cây bao trùm nhỏ nhất.

Thuật toán có thể được cài đặt dựa trên mã giả sau:

Dữ liệu vào: Đồ thị vô hướng liên thông $G = (V_G, E_G)$ với tập đỉnh V_G gồm n đỉnh và tập cạnh E_G gồm m cạnh.

Kết quả: Tập cạnh E_T của cây bao trùm nhỏ nhất của G.

```
Function Prim
{
        Chọn s là một đỉnh bất kỳ của đồ thị.
      Khởi tạo danh sách VT chứa các đỉnh của cây bao trùm nhỏ nhất: VT = {s}
      Khởi tạo danh sách ET chứa các cạnh của cây bao trùm nhỏ nhất: ET = ∅

      Lưu độ dài của cạnh có độ dài nhỏ nhất trong số các cạnh nối đỉnh s với cây
      khung nhỏ nhất đang xây dựng: D[s] = 0
      Lưu đỉnh của cây khung gần s nhất: near[s] = s
      for u ∈ VG \ VH do
      {
            D[u] = c(s, u)
            near[u] = s
      }
      flag = false
      while (flag == false) do
      {
            Tìm đỉnh u ∈ VG \ VH có giá trị D[u] nhỏ nhất
            VT = VT ∪ {u}
            ET = ET ∪ {(u, near[u])}
            if (|VT| == n) then
            {
                  Thông báo T = (VT, ET) là cây bao trùm nhỏ nhất
                  flag = true
            }
            else
            {
                  for v ∈ VG \ VT do
                  {
                        if (D[v] > c[u, v]) then
                        {
                              D[v] = c[u, v]
                              near[v] = u
                        }
                  }
            }
      }
}
```

Ví dụ 9.2: Tìm cây bao trùm nhỏ nhất $\mathbf{T = (V_T, E_T)}$ của đồ thị $G = (V_G, E_G)$ trong hình 9.4.a bằng thuật toán Prim.

1. $V_T = \{1\}$, $V_G - V_T = \{2, 3, 4, 5, 6\}$, $E_T = \emptyset$

near		**D**
$V_G - V_T$	V_T	
2	1	6
3	1	1
4	1	5
5	1	∞
6	1	∞

Chọn đỉnh 3 (đỉnh có giá trị D nhỏ nhất) thêm vào V_T.

Thêm cạnh (1, 3) vào E_T.

2. $V_T = \{1, 3\}$, $V_G - V_T = \{2, 4, 5, 6\}$, $E_T = \{(1, 3)\}$

near		**D**
$V_G - V_T$	V_T	
2	3	5
4	1	5
5	3	6
6	3	4

Chọn đỉnh 6 (đỉnh có giá trị D nhỏ nhất) thêm vào V_T.

Thêm cạnh (3, 6) vào E_T.

3. $V_T = \{1, 3, 6\}$, $V_G - V_T = \{2, 4, 5\}$, $E_T = \{(1, 3), (3, 6)\}$

near		**D**
$V_G - V_T$	V_T	
2	3	5
4	6	2
5	3	6

Chọn đỉnh 4 (đỉnh có giá trị D nhỏ nhất) thêm vào V_T.

Thêm cạnh (6, 4) vào E_T.

4. $V_T = \{1, 3, 6, 4\}$, $V_G - V_T = \{2, 5\}$, $E_T = \{(1, 3), (3, 6), (6, 4)\}$

near		D
$V_G - V_T$	V_T	
2	3	5
5	3	6

Chọn đỉnh 2 (đỉnh có giá trị D nhỏ nhất) thêm vào V_T.

Thêm cạnh (3, 2) vào E_T.

5. $V_T = \{1, 3, 6, 4, 2\}$, $V_G - V_T = \{5\}$, $E_T = \{(1, 3), (3, 6), (6, 4), (3, 2)\}$

near		D
$V_G - V_T$	V_T	
5	2	3

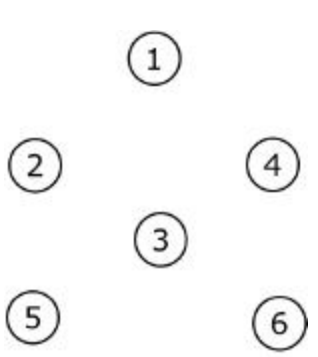
Chọn đỉnh 5 (đỉnh có giá trị D nhỏ nhất) thêm vào V_T.

Thêm cạnh (2, 5) vào E_T.

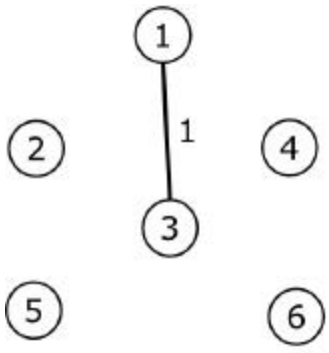
6. $V_T = \{1, 3, 6, 4, 2, 5\}$, $V_G - V_T = \emptyset$, $E_T = \{(1, 3), (3, 6), (6, 4), (3, 2), (2, 5)\}$

Thuật toán dừng do đã xét hết các đỉnh của đồ thị G. Cây bao trùm nhỏ nhất tìm được có tập cạnh $E_T = \{(1, 3), (3, 6), (6, 4), (3, 2), (2, 5)\}$.

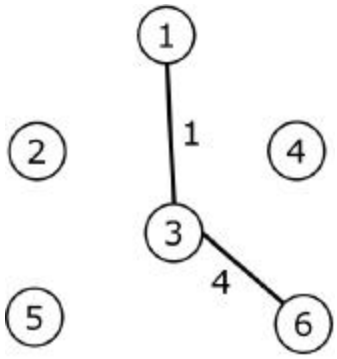
Các bước của thuật toán có thể được minh họa bằng hình ảnh như sau:



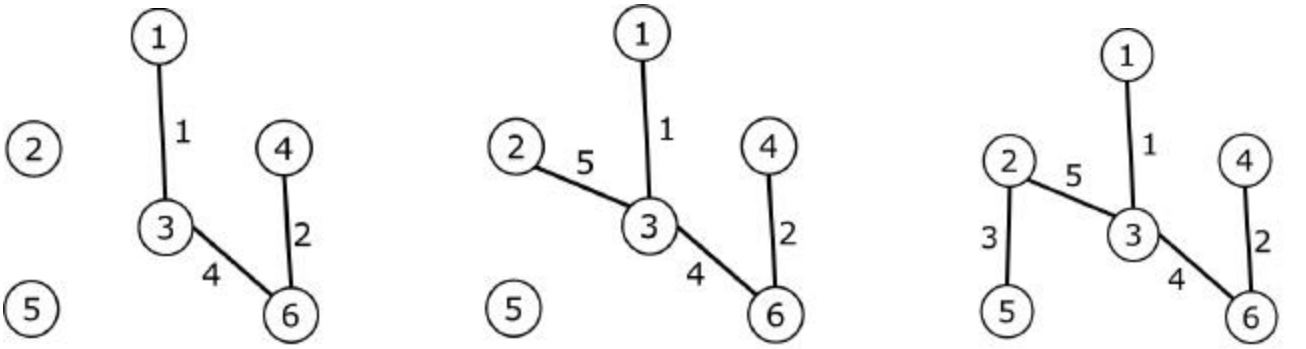
1. $V_T = \{1\}$



2. $V_T = \{1, 3\}$



3. $V_T = \{1, 3, 6\}$



4. $V_T = \{1, 3, 6, 4\}$ 5. $V_T = \{1, 3, 6, 4, 2\}$ 6. $V_T = \{1, 3, 6, 4, 2, 5\}$

Hình 9.6: Minh họa của giải thuật Prim

Ngoài việc tìm cây bao trùm nhỏ nhất, hai thuật toán Kruskal và Prim đều có thể chỉnh sửa để tìm cây bao trùm có trọng số lớn nhất bằng cách lần lượt chọn các cạnh có trọng số lớn nhất thay vì chọn cạnh có trọng số nhỏ nhất. Cạnh được chọn không tạo thành chu trình với các cạnh đã có.

TÓM TẮT

Trong bài này, học viên cần nắm:

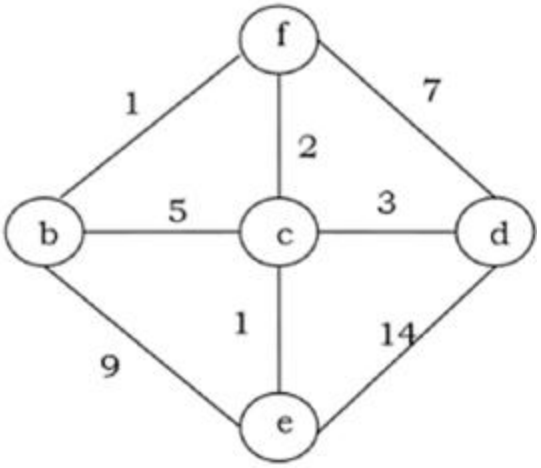
Cây bao trùm của đồ thị G với n đỉnh (n ≥ 2) là đồ thị bộ phận của G và có tính liên thông. Cây bao trùm nhỏ nhất của G là cây bao trùm có tổng trọng số của các cạnh là bé nhất.

Cây bao trùm nhỏ nhất có thể được xác định bằng cách sử dụng giải thuật Kruskal và giải thuật Prim. Hai giải thuật này dựa trên chiến lược tham lam để lựa chọn cạnh để xây dựng cây bao trùm nhỏ nhất.

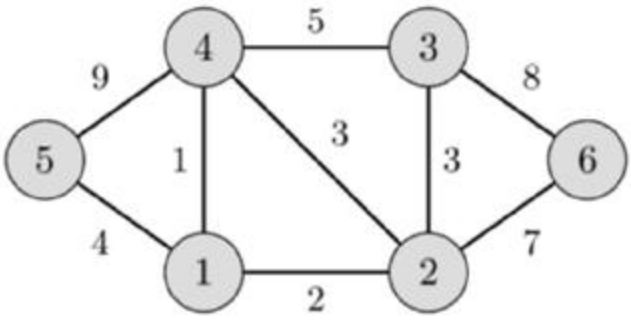
CÂU HỎI ÔN TẬP

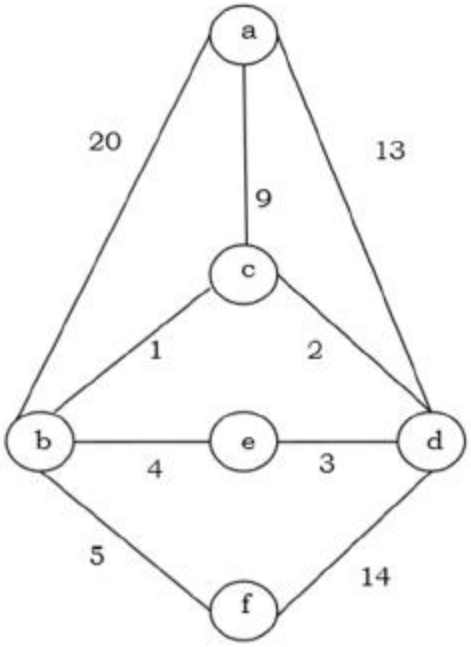
Câu 1: Tìm cây bao trùm nhỏ nhất của các đồ thị sau bằng thuật toán Kruskal.

a.



b.





c.

Câu 2: Tìm cây bao trùm nhỏ nhất của các đồ thị cho tại câu 1 bằng thuật toán Prim.

Câu 3: Viết hàm tìm cây bao trùm nhỏ nhất bằng thuật toán Kruskal.

Câu 4: Viết hàm tìm cây bao trùm nhỏ nhất bằng thuật toán Prim.

BÀI 10: ĐƯỜNG ĐI NGẮN NHẤT

Sau khi học xong bài này, học viên có thể:

- *Hiểu được các giải thuật tìm đường đi ngắn nhất giữa 2 đỉnh trên đồ thị, bao gồm: thuật toán Dijstra, thuật toán Bellman-Ford, và thuật toán Floyd.*
- *Vận dụng được các thuật toán tìm đường đi ngắn nhất để giải các bài toán cụ thể.*

10.1 BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

Cho đồ thị G = (V, E) và hai đỉnh v, w ∈ V. Bài toán đặt ra ở đây là tìm đường đi ngắn nhất (nếu có) từ đỉnh v đến đỉnh w. Giải thuật duyệt đồ thị theo chiều rộng đã giúp chúng ta giải được bài toán này trong trường hợp đồ thị G không có trọng số. Tuy nhiên, nếu đồ thị G đã cho là đồ thị có trọng số, giải thuật tìm kiếm theo chiều rộng không thể áp dụng để giải bài toán này.

Bài toán tìm đường đi ngắn nhất trong đồ thị có trọng số được định nghĩa như sau: Cho đồ thị có trọng số (G, c), trong đó G = (V, E) với V là tập đỉnh, E là tập cung, c là tập các trọng số của các cạnh của G. Hãy tìm đường đi có tổng trọng số bé nhất (nếu có) từ đỉnh s (∈ V) tới tất cả các đỉnh còn lại của G.

Độ dài đường đi ngắn nhất từ đỉnh s tới đỉnh g được gọi là khoảng cách từ s tới g. Nếu không có đường đi, khoảng cách là ∞.

10.2 THUẬT TOÁN DIJKSTRA

Thuật toán Dijkstra để tìm đường đi ngắn nhất từ một đỉnh s tới tất cả các đỉnh còn lại trong đồ thị G = (V, E) được phát biểu như sau:

1. Khởi tạo danh sách liên kết S = ∅
2. Khởi tạo danh sách liên kết D chứa khoảng cách từ đỉnh bắt đầu s tới tất cả các đỉnh của đồ thị. Tại thời điểm ban đầu, D[i] = ∞, trong đó i ∈ V, và D[s] = 0.

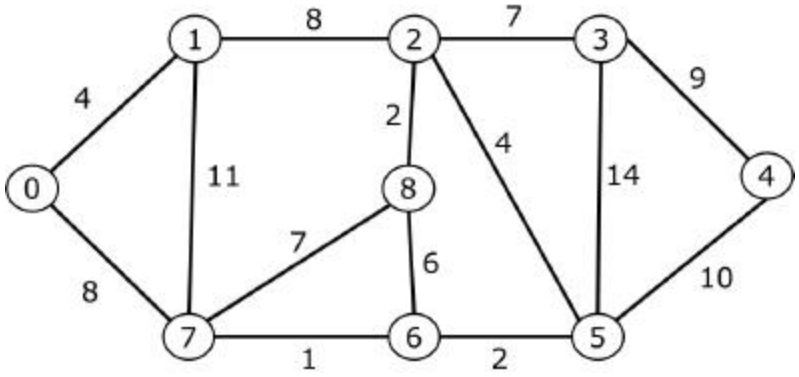
3. Nếu S chưa bao gồm tất cả các đỉnh của đồ thị thì:

a. Chọn một đỉnh u ∉ S và có giá trị nhỏ nhất trong danh sách D.

b. Thêm u vào danh sách S.

c. Cập nhật khoảng cách của các đỉnh liền kề với u trong danh sách D như sau: với mỗi đỉnh v liền kề với u, nếu D[u] + c(u, v) nhỏ hơn D[v] thì cập nhật lại D[v] = D[u] + c(u, v).

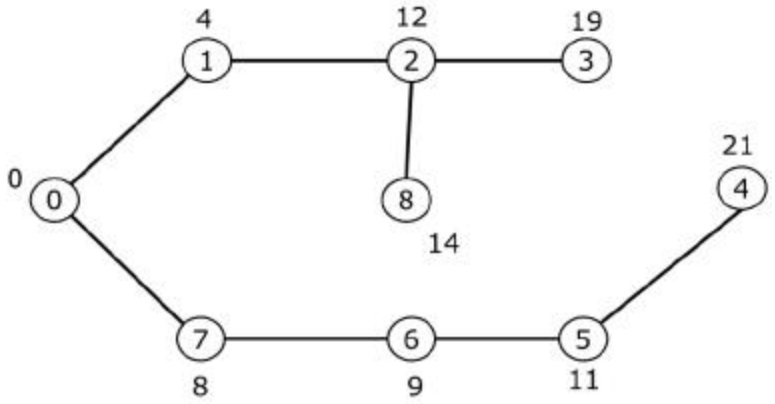
Ví dụ 10.1: Cho đồ thị trong hình 10.1. Áp dụng thuật toán Dijstra tìm đường đi ngắn nhất từ đỉnh 0 tới tất cả các đỉnh còn lại.



Hình 10.1: Ví dụ minh họa giải thuật Dijstra

1. S = ∅, D = {0, ∞, ∞, ∞, ∞, ∞, ∞, ∞, ∞}
2. Chọn u = 0 → S = {0}, D = {0, 4, ∞, ∞, ∞, ∞, ∞, 8, ∞}.
3. Chọn u = 1 → S = {0, 1}, D = {0, 4, 12, ∞, ∞, ∞, ∞, 8, ∞}.
4. Chọn u = 7 → S = {0, 1, 7}, D = {0, 4, 12, ∞, ∞, ∞, 9, 8, 15}.
5. Chọn u = 6 → S = {0, 1, 7, 6}, D = {0, 4, 12, ∞, ∞, 11, 9, 8, 15}.
6. Chọn u = 5 → S = {0, 1, 7, 6, 5}, D = {0, 4, 12, ∞, 21, 11, 9, 8, 15}.
7. Chọn u = 2 → S = {0, 1, 7, 6, 5, 2}, D = {0, 4, 12, 19, 21, 11, 9, 8, 14}.
8. Chọn u = 8 → S = {0, 1, 7, 6, 5, 2, 8}, D = {0, 4, 12, 19, 21, 11, 9, 8, 14}.
9. Chọn u = 3 → S = {0, 1, 7, 6, 5, 2, 8}, D = {0, 4, 12, 19, 21, 11, 9, 8, 14}.
10. Chọn u = 4 → S = {0, 1, 7, 6, 5, 2, 8, 4}, D = {0, 4, 12, 19, 21, 11, 9, 8, 14}.

Thuật toán dừng vì tập S đã bao gồm tất cả các đỉnh của đồ thị. Đường đi ngắn nhất từ 0 tới tất cả các đỉnh của đồ thị được thể hiện trên đồ thị trong hình 10.2. Số nằm bên cạnh mỗi đỉnh thể hiện tổng trọng số từ đỉnh 0 tới đỉnh đang xét.



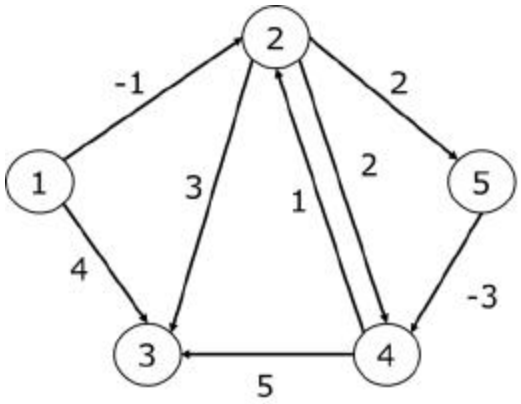
Hình 10.2: Cây biểu diễn đường đi ngắn nhất từ 0 tới tất cả các đỉnh của đồ thị trong hình 10.1

10.3 THUẬT TOÁN BELLMAN- FORD

Giải thuật Dijstra không hoạt động được trong trường hợp đồ thị có trọng số âm. Do vậy, trong trường hợp tổng quát, giải thuật Bellman-Ford được sử dụng để giải bài toán tìm đường đi ngắn nhất.

1. Khởi tạo danh sách liên kết D chứa khoảng cách từ đỉnh bắt đầu s tới tất cả các đỉnh của đồ thị. Tại thời điểm ban đầu, D[i] = ∞, trong đó i ∈ V, và D[s] = 0.
2. Với mỗi cạnh (u, v), nếu D[u] + c(u, v) < D[v] thì cập nhật lại D[v] = D[u] + c(u, v). Lặp lại bước này n – 1 lần, trong đó n là số đỉnh của đồ thị.

Ví dụ 10.2: Cho đồ thị trong hình 10.3. Sử dụng giải thuật Bellman-Ford để tìm đường đi ngắn nhất tới tất cả các đỉnh từ đỉnh 1.



Hình 10.3: Ví dụ minh họa giải thuật Bellman-Ford

Các bước cập nhật mảng D của thuật toán được thể hiện trong bảng dưới đây. Mỗi dòng trong bảng ứng với giá trị của mảng D sau mỗi lần cập nhật. Việc cập nhật D

được thực hiện 4 lần do số đỉnh của đồ thị là 5. Giả sử thứ tự xét các cạnh như sau: (2, 5), (4, 2), (2, 4), (1, 2), (1, 3), (4, 3), (2, 3), (5, 4).

Khởi tạo: D = {0, ∞, ∞, ∞, ∞}

1. Lần lặp 1:

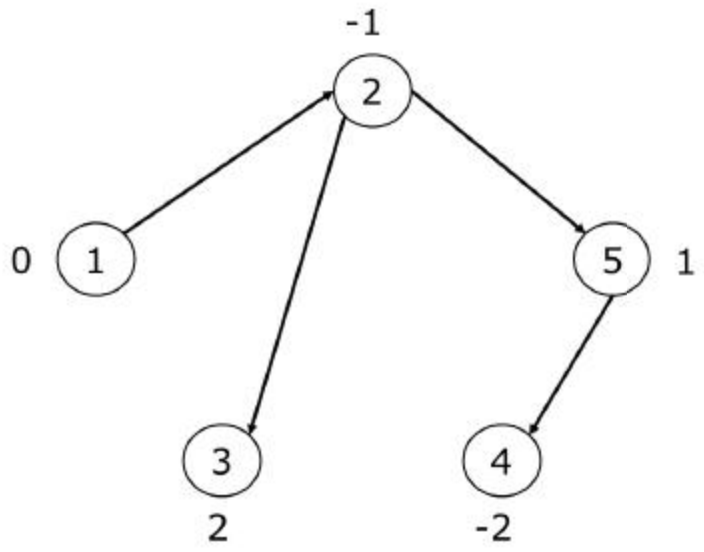
- Xét cạnh (2, 5) → D = {0, ∞, ∞, ∞, ∞}
- Xét cạnh (4, 2) → D = {0, ∞, ∞, ∞, ∞}
- Xét cạnh (2, 4) → D = {0, ∞, ∞, ∞, ∞}
- Xét cạnh (1, 2) → D = {0, -1, ∞, ∞, ∞}
- Xét cạnh (1, 3) → D = {0, -1, 4, ∞, ∞}
- Xét cạnh (4, 3) → D = {0, -1, 4, ∞, ∞}
- Xét cạnh (2, 3) → D = {0, -1, 2, ∞, ∞}
- Xét cạnh (5, 4) → D = {0, -1, 2, ∞, ∞}

2. Lần lặp 2:

- Xét cạnh (2, 5) → D = {0, -1, 2, ∞, 1}
- Xét cạnh (4, 2) → D = {0, -1, 2, ∞, 1}
- Xét cạnh (2, 4) → D = {0, -1, 2, 1, 1}
- Xét cạnh (1, 2) → D = {0, -1, 2, 1, 1}
- Xét cạnh (1, 3) → D = {0, -1, 2, 1, 1}
- Xét cạnh (4, 3) → D = {0, -1, 2, 1, 1}
- Xét cạnh (2, 3) → D = {0, -1, 2, 1, 1}
- Xét cạnh (5, 4) → D = {0, -1, 2, -2, 1}

3. Lần lặp 3 và 4 được thực hiện tương tự. Tuy nhiên, giá trị của mảng D không được cập nhật lại do D đã tối ưu.

Thuật toán dừng vì đã lặp đủ số lần theo yêu cầu. Đường đi ngắn nhất từ 1 tới tất cả các đỉnh của đồ thị được thể hiện trên đồ thị trong hình 10.4. Số nằm bên cạnh mỗi đỉnh thể hiện tổng trọng số từ đỉnh 0 tới đỉnh đang xét.



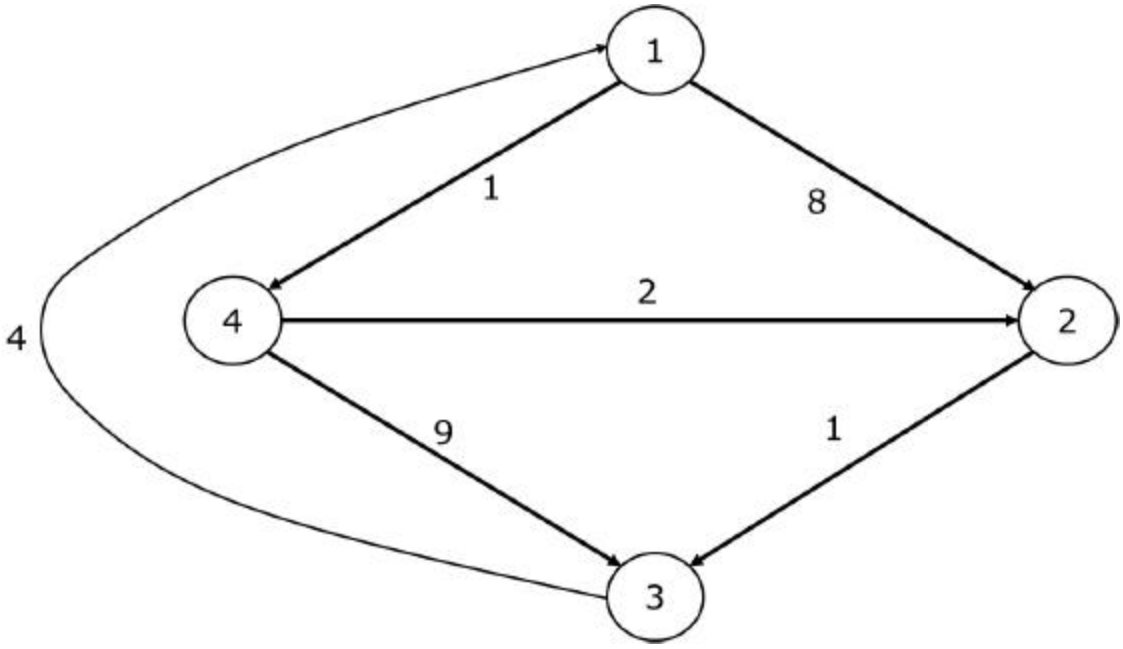
Hình 10.4: Bụi biểu diễn đường đi ngắn nhất từ 1 tới tất cả các đỉnh của đồ thị trong hình 10.3

10.4 THUẬT TOÁN FLOYD

Tương tự giải thuật Dijsktra và Bellman-Ford, thuật toán Floyd (còn gọi là thuật toán Floyd-Warshall) giúp chúng ta giải bài toán tìm đường đi ngắn nhất. Tuy nhiên, khác với Dijstra và Bellman-Ford, thuật toán Floyd tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh trong đồ thị. Thuật toán Floyd gồm các bước như sau:

1. Xóa khuyên và các cạnh song song nếu có. Trong số các cạnh song song, giữ lại cạnh có trọng số nhỏ nhất.
2. Khởi tạo ma trận D = ma trận kề của đồ thị đã cho.
3. Cập nhật ma trận D bằng cách coi tất cả các đỉnh của đồ thị là một đỉnh trung gian theo nguyên tắc sau: xét k là một đỉnh bất kỳ của đồ thị, với mọi cặp (i, j), trong đó i là đỉnh nguồn và j là đỉnh đích, nếu k là đỉnh trung gian trên đường đi từ i tới j và D[i][j]> D[i][k] + D[k][j] thì cập nhật D[i][j] = D[i][k] + D[k][j].

Ví dụ 10.3: Cho đồ thị trong hình 10.5. Sử dụng thuật Floyd tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh trong đồ thị.



Hình 10.5: Ví dụ minh họa giải thuật Floyd

1. Khởi tạo ma trận D:

$$D = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix}$$

2. Xét đỉnh trung gian k = 1, cập nhật lại giá trị của ma trận D:

$$D = \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix}$$

3. Xét đỉnh trung gian k = 2, cập nhật lại giá trị của ma trận D:

$$D = \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix}$$

4. Xét đỉnh trung gian k = 3, cập nhật lại giá trị của ma trận D:

$$D = \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

5. Xét đỉnh trung gian k = 4, cập nhật lại giá trị của ma trận D:

$$D = \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix}$$

Thuật toán dừng sau khi xét tất cả các đỉnh của đồ thị. Ma trận D thu được thể hiện độ dài của đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị.

TÓM TẮT

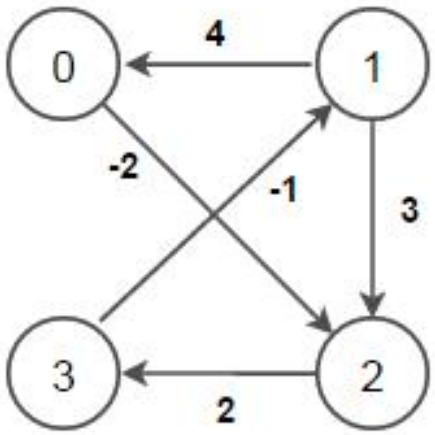
Trong bài này, học viên cần nắm:

Bài toán tìm đường đi ngắn nhất trong đồ thị có trọng số được định nghĩa như sau: Cho đồ thị có trọng số, hãy tìm đường đi có tổng trọng số bé nhất (nếu có) từ đỉnh s ($\in V$) tới tất cả các đỉnh còn lại của G. Độ dài đường đi ngắn nhất từ đỉnh s tới đỉnh g được gọi là khoảng cách từ s tới g. Nếu không có đường đi, khoảng cách là ∞.

Bài toán tìm đường đi ngắn nhất có thể giải được bằng cách áp dụng các thuật toán Dijstra, Bellman-Ford, và Floyd. Trong đó, giải thuật Dijstra và giải thuật Bellman-Ford được dùng để tìm đường đi từ một đỉnh tới các đỉnh còn lại của đồ thị. Và giải thuật Dijstra chỉ hoạt động được trong trường hợp đồ thị không có trọng số âm. Nếu đồ thị có trọng số âm, giải thuật Bellman-Ford được sử dụng. Giải thuật Floyd giúp chúng ta tìm được đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị.

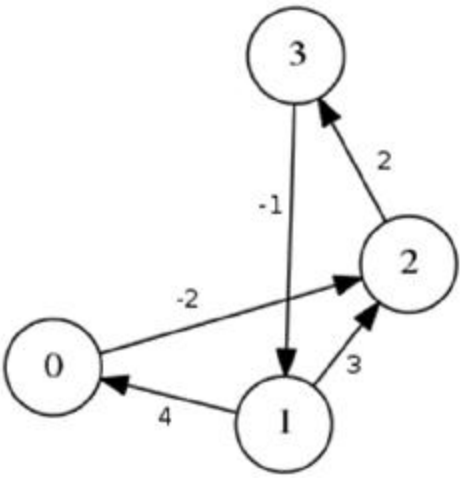
CÂU HỎI ÔN TẬP

Câu 1: Áp dụng giải thuật Dijstra tìm đường đi ngắn nhất từ đỉnh có chỉ số nhỏ nhất tới các đỉnh còn lại của các đồ thị sau:

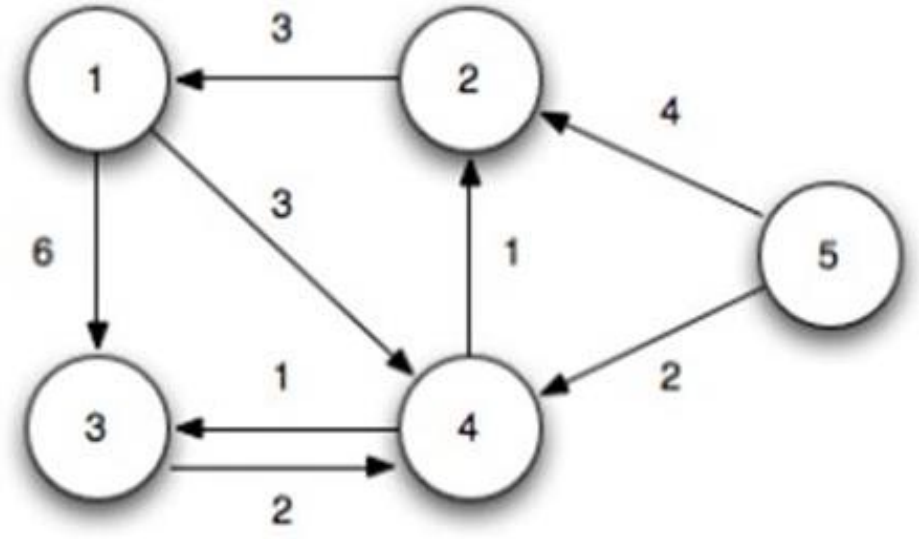


a.

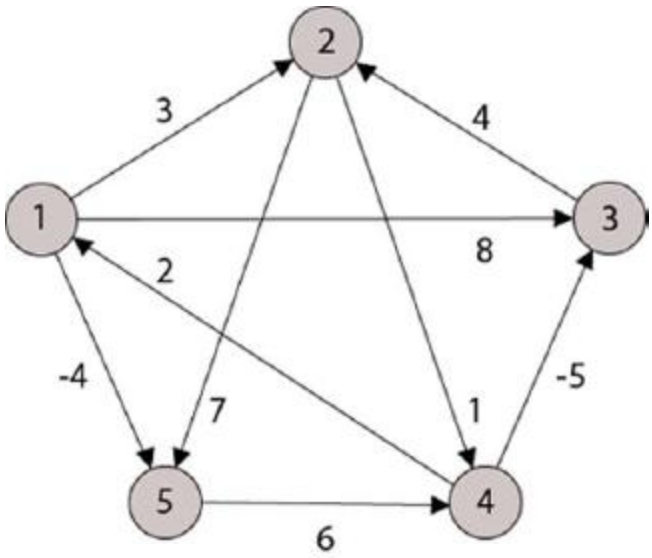
b.



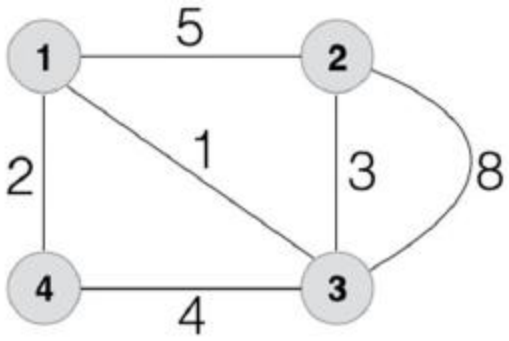
c.



d.



e.



Câu 2: Áp dụng giải thuật Bellman-Ford tìm đường đi ngắn nhất từ đỉnh có chỉ số nhỏ nhất tới các đỉnh còn lại của các đồ thị trong bài 1.

Câu 3: Viết hàm tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh còn lại sử dụng giải thuật Dijstra.

Câu 4: Viết hàm tìm đường đi ngắn nhất từ một đỉnh tới các đỉnh còn lại sử dụng giải thuật Bellman-Floyd.

Câu 5: Viết hàm tìm đường đi ngắn nhất giữa tất cả các cặp đỉnh của đồ thị sử dụng giải thuật Floyd.

TÀI LIỆU THAM KHẢO

1. Dương Anh Đức, Trần Hạnh Nhi (2000). *Cấu trúc dữ liệu & thuật toán*. Đại học Khoa học Tự nhiên, Tp. Hồ Chí Minh.
2. Đặng Huy Ruận (2000). *Lý thuyết đồ thị và ứng dụng*. Nhà xuất bản Khoa học và Kỹ thuật, Hà Nội.
3. Hoàng Chí Thành (2007). *Đồ thị và các thuật toán*. Nhà xuất bản giáo dục.
4. Nguyễn Trung Trực (1992). *Cấu trúc dữ liệu*. Khoa CNTT, Trường ĐH Bách Khoa TP.HCM.
5. Lê Minh Hoàng. (1999 - 2002). *Giải thuật & lập trình*. Đại học Sư Phạm Hà Nội.
6. Robin J. Wilson (1996). *Introduction to Graph Theory, fourth edition.* Addison Wesley Longman Limited.
7. Richard Neapolitan and Kumarss (2004). *Foundations of Algorithms Using C++ Pseudocode*. Jones and Bartlett Publishers.
8. Nguyễn Văn Linh (2003). *Giải thuật*. Đại học Cần Thơ.
9. Nguyễn Hà Giang (2008). *Bài giảng Cấu trúc dữ liệu và Giải thuật*. Đại học Kỹ thuật Công nghệ, Tp. Hồ Chí Minh.
10. ThS. Văn Thị Thiên Trang, Cấu trúc dữ liệu và giải thuật (2019). Đại học Kỹ thuật Công nghệ, Tp. Hồ Chí Minh.