Chapter 7. Trees

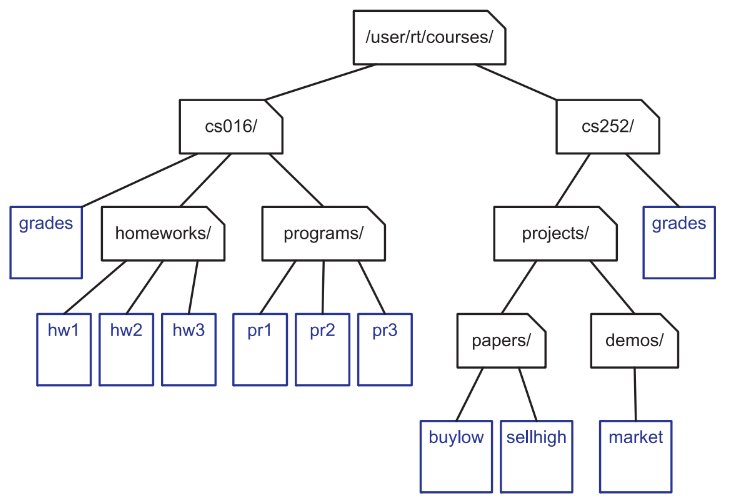
7.1 General Trees

Các chuyên gia về năng suất cho rằng sự đột phá đến từ tư duy "phi tuyến tính." Trong chương này, chúng ta thảo luận về một cấu trúc dữ liệu quan trọng nhất trong máy tính - cây. Cấu trúc cây là một đột phá trong tổ chức dữ liệu, giúp triển khai thuật toán nhanh chóng hơn so với cấu trúc tuyến tính. Cây cũng tổ chức dữ liệu một cách tự nhiên và xuất hiện rộng rãi trong hệ thống tập tin, giao diện đồ họa, cơ sở dữ liệu, trang web, và hệ thống máy tính khác.

Khi nói về tư duy "phi tuyến tính," chúng ta ám chỉ mối quan hệ tổ chức phong phú hơn so với mối quan hệ "trước" và "sau" trong các chuỗi. Mối quan hệ trong cây có tính phân cấp, với một số đối tượng ở "trên" và một số ở "dưới." Thuật ngữ chủ yếu của cây đến từ cây gia đình, với "cha mẹ," "con cái," "tổ tiên," và "hậu duệ" là những từ phổ biến để mô tả mối quan hệ. Xem ví dụ cây gia đình trong Hình 7.1.

7.1.1 Định nghĩa và tính chất của cây

Một cây là một loại dữ liệu trừu tượng lưu trữ các phần tử theo cấp bậc. Trừ phần tử đầu tiên (gốc), mỗi phần tử trong cây đều có một phần tử cha và không hoặc nhiều hơn một phần tử con. Thông thường, cây được biểu diễn bằng cách đặt các phần tử trong hình oval hoặc hình chữ nhật và vẽ các kết nối giữa cha và con bằng các đường thẳng. (Xem Hình 7.2.) Chúng ta thường gọi phần tử đầu tiên là gốc của cây, nhưng nó được vẽ là phần tử cao nhất, với các phần tử khác được kết nối phía dưới (ngược lại so với cây thực vật).



Một cạnh của cây T là một cặp nút (u, v) sao cho u là cha của v, hoặc ngược lại. Một đường đi của T là một chuỗi nút sao cho bất kỳ hai nút liên tiếp nào trong chuỗi đều tạo thành một cạnh.

Cây được xếp thứ tự nếu có một thứ tự tuyến tính được xác định cho các con của mỗi nút; nghĩa là, chúng ta có thể xác định các con của một nút là con đầu tiên, con thứ hai, con thứ ba, và cứ thế. Thứ tự này được xác định bởi cách cây được sử dụng, và thường được chỉ định bằng cách vẽ cây với các anh chị em được sắp xếp từ trái sang phải, tương ứng với mối quan hệ tuyến tính của chúng. Các cây được xếp thứ tự thường chỉ định mối quan hệ thứ tự tuyến tính giữa các anh chị em bằng cách liệt kê chúng theo thứ tự đúng.

7.1.2 Chức năng cây

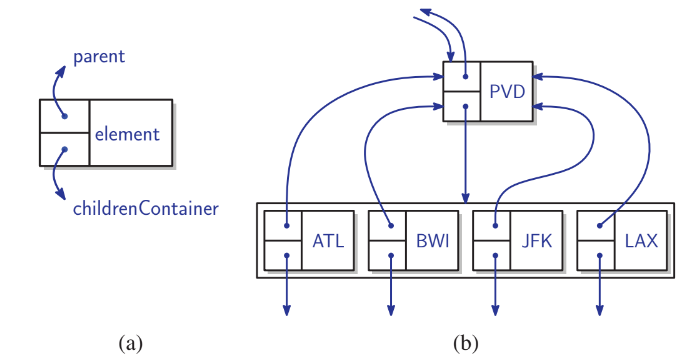
Cây là một kiểu dữ liệu trừu tượng lưu trữ phần tử ở các nút. Mỗi nút có một đối tượng vị trí để truy cập công khai. Chức năng của cây bao gồm kích thước, trống, nút gốc, và danh sách vị trí. Vị trí cây cung cấp quyền truy cập đến các nút lá và các nút liên quan. Các chức năng cơ bản bao gồm **parent()**, **children()**, **isRoot()**, và **isExternal()**. Cây cũng cung cấp các hàm **size()**, **empty()**, **root()**, và **positions()**. Các chức năng cập nhật đặc biệt không được định nghĩa ở đây và sẽ được thảo luận trong các ứng dụng cụ thể.

7.1.4 A Linked Structure for General Trees

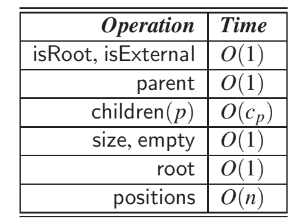
Cách tự nhiên để thực hiện một cây T là sử dụng một cấu trúc liên kết, trong đó chúng ta biểu diễn mỗi nút của T bằng một đối tượng vị trí p (xem Hình 7.5(a)) với các trường sau:

* Một tham chiếu đến phần tử của nút,
* Một liên kết đến nút cha,
* Và một loại bộ sưu tập (ví dụ, một danh sách hoặc mảng) để lưu trữ liên kết đến các nút con. Nếu p là gốc của T, thì trường cha của p là NULL.

Ngoài ra, chúng ta lưu một tham chiếu đến gốc của T và số lượng nút của T trong các biến nội tại.



Bảng 7.1 tóm tắt hiệu suất của việc triển khai cây bằng cấu trúc liên kết. Phân tích để lại làm bài tập (C-7.27), nhưng chúng tôi lưu ý rằng, bằng cách sử dụng một bộ chứa để lưu trữ các con của mỗi nút p, chúng ta có thể triển khai hàm children(p) bằng cách sử dụng trình lặp cho bộ chứa để liệt kê các phần tử của nó.



7.2 Thuật toán Traversal cây

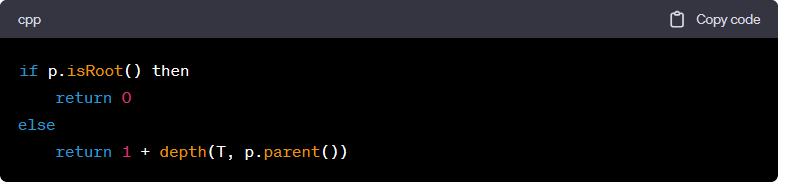
7.2.1 Depth and Height

Để p là một nút của cây T. Độ sâu của p là số lượng tổ tiên của p, không tính p chính nó. Ví dụ, trong cây của Hình 7.2, nút chứa International có độ sâu là 2. Lưu ý rằng định nghĩa này ngụ ý rằng độ sâu của nút gốc của T là 0. Độ sâu của nút p cũng có thể được định nghĩa đệ quy như sau:

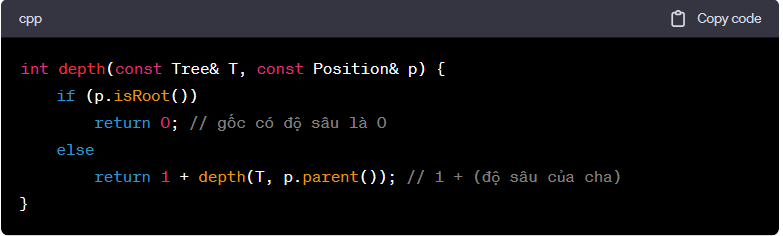
* Nếu p là gốc, thì độ sâu của p là 0.
* Ngược lại, độ sâu của p là một cộng với độ sâu của cha của p.

Dựa trên định nghĩa trên, thuật toán đệ quy depth(T, p) được hiển thị trong Đoạn mã 7.3 tính độ sâu của một nút được tham chiếu bởi vị trí p của T bằng cách gọi chính nó đệ quy trên cha của p và cộng thêm 1 vào giá trị trả về.

Thuật toán depth(T, p):



Đoạn mã 7.4 là một triển khai C++ đơn giản của thuật toán depth.



Thời gian chạy của thuật toán depth(T, p) là O(dp), trong đó dp là độ sâu của nút p trong cây T, vì thuật toán thực hiện một bước đệ quy có thời gian hằng số cho mỗi tổ tiên của p. Do đó, trong trường hợp xấu nhất, thuật toán độ sâu chạy trong thời gian O(n), trong đó n là tổng số nút trong cây T, vì một số nút có thể có độ sâu như vậy trong T. Mặc dù thời gian chạy như vậy là một hàm của kích thước đầu vào, nhưng nó chính xác hơn khi mô tả thời gian chạy dựa trên tham số dp, vì đôi khi dp thường nhỏ hơn n nhiều.

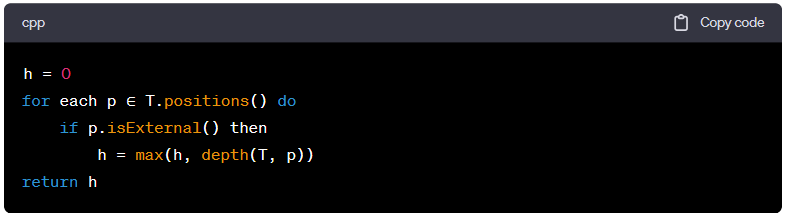
Chiều cao của một nút p trong cây T cũng được định nghĩa đệ quy:

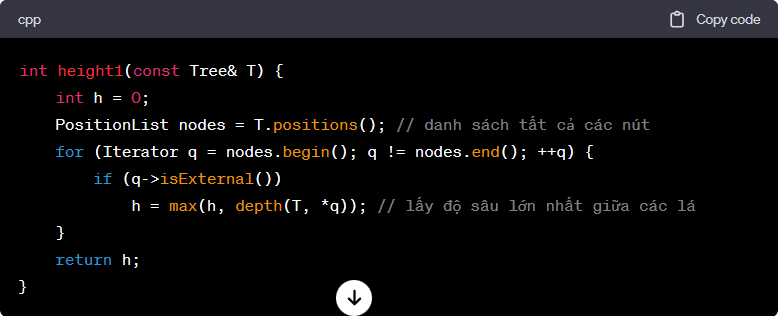
* Nếu p là nút lá, thì chiều cao của p là 0.
* Ngược lại, chiều cao của p là một cộng với chiều cao lớn nhất của một con của p.

Chiều cao của cây T là chiều cao của gốc của T. Ví dụ, cây của Hình 7.2 có chiều cao là 4. Ngoài ra, chiều cao cũng có thể được xem xét như sau.

Mệnh đề 7.4: Chiều cao của một cây bằng với độ sâu tối đa của các nút lá của nó.

Chúng tôi để lại chứng minh cho sự thật này làm bài tập (R-7.7). Dựa trên mệnh đề này, chúng tôi trình bày một thuật toán, height1, để tính chiều cao của một cây T. Nó được hiển thị trong Đoạn mã 7.5. Nó liệt kê tất cả các nút trong cây và gọi hàm depth (Đoạn mã 7.3) để tính độ sâu của mỗi nút lá.

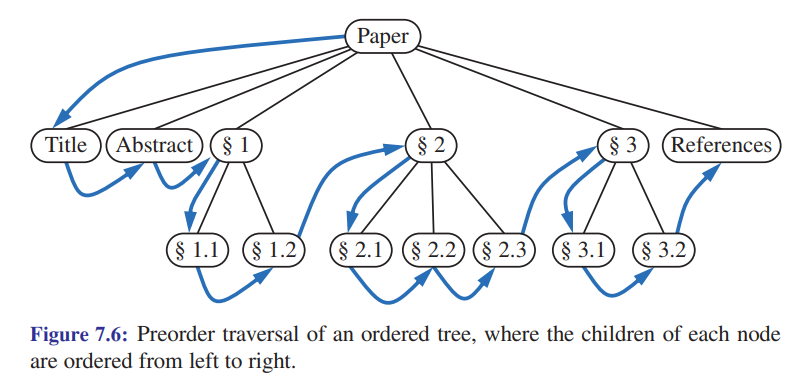
Thuật toán height1(T):  Đoạn mã 7.6 là một triển khai C++ đơn giản của thuật toán này.

cpp  


Thật không may, thuật toán height1 không hiệu quả. Vì height1 gọi thuật toán depth(p) cho mỗi nút lá p của T, thời gian chạy của height1 được cho bởi O(n+∑p(1+dp)), trong đó n là số lượng nút của T, dp là độ sâu của nút p, và E là tập hợp các nút lá của T. Trong trường hợp xấu nhất, tổng ∑p(1+ dp) tỷ lệ thuận với n^2. Do đó, thuật toán height1 chạy trong thời gian O(n^2).

7.2.2 Preorder Traversal

Duyệt theo thứ tự trước của cây T thăm nút gốc trước và sau đó duyệt các cây con đệ quy. Nếu cây được sắp xếp, thì thứ tự duyệt sẽ tuân theo thứ tự của các con. Cụ thể hành động thăm nút phụ thuộc vào ứng dụng và có thể là bất kỳ điều gì từ việc tăng một bộ đếm đến thực hiện một tính toán phức tạp. Mã giả của duyệt theo thứ tự trước được thể hiện trong Đoạn mã 7.9.



Duyệt theo thứ tự trước của cây T thăm nút gốc trước, sau đó duyệt cây con đệ quy. Thuật toán này hiệu quả với thời gian chạy O(n), với n là số nút trong cây. Đoạn mã C++ in các phần tử của cây theo thứ tự trước.

Thuật toán **preorderPrint(T, p)**, được triển khai trong C++ trong Đoạn mã 7.10, thực hiện việc in theo thứ tự trước của cây con có gốc tại nút p của T, nghĩa là thực hiện duyệt theo thứ tự trước của cây con có gốc tại p và in phần tử được lưu trữ tại một nút khi nút đó được thăm. Lưu ý rằng, đối với một cây được sắp xếp T, hàm **T.children(p)** trả về một trình lặp truy cập các con của p theo thứ tự. Chúng ta giả sử rằng Iterator là kiểu trình lặp này. Cho một trình lặp q, vị trí liên kết với nó được đưa ra bởi \*q.

**void preorderPrint(const Tree& T, const Position& p) {**

**cout << \*p; // print element**

**PositionList ch = p.children(); // list of children for**

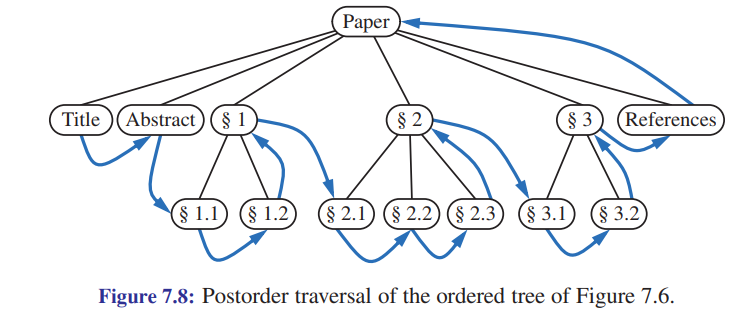
**(Iterator q = ch.begin(); q != ch.end(); ++q)**

**{ cout << " ";**

**preorderPrint(T, \*q); } }**

7.2.3 Postorder Traversal

Một thuật toán duyệt cây quan trọng khác là duyệt theo thứ tự sau (postorder traversal). Thuật toán này có thể được coi như ngược lại của duyệt theo thứ tự trước, vì nó đệ quy duyệt các cây con có gốc ở các con của gốc trước, sau đó thăm gốc. Tuy nhiên, nó giống với duyệt theo thứ tự trước ở chỗ chúng ta sử dụng nó để giải quyết một vấn đề cụ thể bằng cách chuyên biệt hóa một hành động liên quan đến "thăm" một nút p. Tuy nhiên, giống như duyệt theo thứ tự trước, nếu cây được sắp xếp, chúng ta thực hiện các cuộc gọi đệ quy cho các con của một nút p theo thứ tự được xác định. Mã giả cho duyệt theo thứ tự sau được đưa ra trong Đoạn mã 7.8.

****

7.3 Binary Trees

7.3.1 The Binary Tree ADT

Trong phần này, chúng ta giới thiệu một kiểu dữ liệu trừu tượng cho một cây nhị phân. Tương tự như với ADT cây trước đó, mỗi nút của cây lưu trữ một phần tử và liên kết với một đối tượng vị trí, cung cấp quyền truy cập công khai đến các nút. Bằng cách quá tải toán tử giải tham chiếu, phần tử liên kết với một vị trí p có thể được truy cập bằng cách sử dụng \*p.

Ngoài ra, một vị trí p hỗ trợ các thao tác sau:

* **p.left()**: Trả về nút con bên trái của p; một điều kiện lỗi xảy ra nếu p là một nút lá.
* **p.right()**: Trả về nút con bên phải của p; một điều kiện lỗi xảy ra nếu p là một nút lá.
* **p.parent()**: Trả về nút cha của p; một lỗi xảy ra nếu p là gốc.
* **p.isRoot()**: Trả về true nếu p là gốc và false nếu ngược lại.
* **p.isExternal()**: Trả về true nếu p là nút lá và false nếu ngược lại.

Cây chính cung cấp các thao tác giống như ADT cây tiêu chuẩn. Nhớ rằng một danh sách vị trí là một danh sách các vị trí cây.

* **size()**: Trả về số lượng nút trong cây.
* **empty()**: Trả về true nếu cây trống và false nếu ngược lại.
* **root()**: Trả về một vị trí cho gốc của cây; một lỗi xảy ra nếu cây trống.
* **positions()**: Trả về một danh sách vị trí của tất cả các nút trong cây.

7.3.3 Properties of Binary Trees

Cây nhị phân có một số thuộc tính thú vị liên quan đến mối quan hệ giữa chiều cao và số nút của chúng. Chúng ta ký hiệu tập hợp tất cả các nút của một cây T, cùng ở cùng một độ sâu d, là cấp d của T. Trong một cây nhị phân, cấp 0 có một nút (gốc), cấp 1 có tối đa hai nút (con của gốc), cấp 2 có tối đa bốn nút, và cứ như vậy. (Xem Hình 7.12.) Nói chung, cấp d có tối đa 2^d nút.

Chúng ta có thể thấy rằng số lượng tối đa nút trên các cấp của một cây nhị phân tăng một cách mũi mẫu khi chúng ta đi xuống cây. Từ quan sát đơn giản này, chúng ta có thể suy ra các thuộc tính sau liên quan đến chiều cao của một cây nhị phân và số nút của nó. Một giải thích chi tiết về những thuộc tính này được để lại làm bài tập.

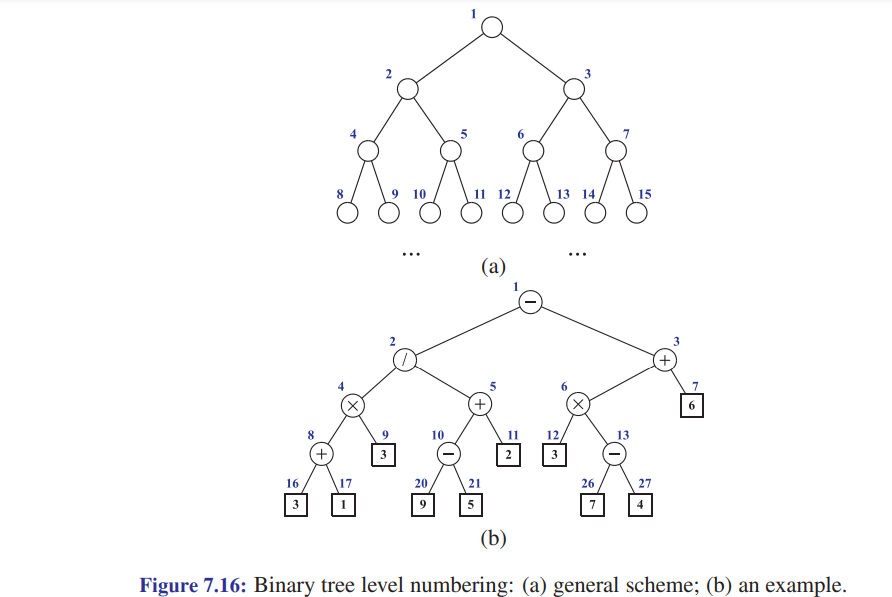
7.3.4 A Linked Structure for Binary TreesTop of Form

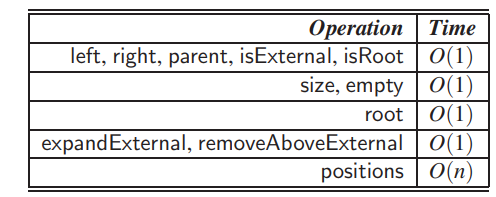
Ở phần này, chúng ta trình bày một cách triển khai của cây nhị phân T dưới dạng một cấu trúc liên kết, được gọi là **LinkedBinaryTree**. Chúng ta đại diện cho mỗi nút **v** của T bằng một đối tượng nút lưu trữ phần tử liên quan và các con trỏ đến nút cha và hai nút con của nó. (Xem Hình 7.13.) Đơn giản hóa, chúng ta giả sử cây là đầy đủ, có nghĩa là mỗi nút có hoặc không có hai nút con.

Trong Hình 7.14, chúng ta thể hiện một biểu diễn cấu trúc liên kết của một cây nhị phân. Cấu trúc này lưu trữ kích thước của cây, tức là số nút trong cây, và một con trỏ đến gốc của cây. Phần còn lại của cấu trúc bao gồm các nút được liên kết với nhau một cách phù hợp. Nếu **v** là gốc của T, thì con trỏ đến nút cha là NULL, và nếu **v** là một nút lá, thì các con trỏ đến các nút con của **v** là NULL.

7.3.5 A Vector-Based Structure for Binary Trees

Một cách đơn giản để biểu diễn cây nhị phân là sử dụng một vector S với hàm đánh số cấp f. Mỗi nút v của cây được liên kết với phần tử tại chỉ số f(v) trong vector. Việc này giúp thực hiện các thao tác trên cây một cách dễ dàng và hiệu quả, nhưng có thể tốn không gian đáng kể nếu cây có chiều cao lớn. Các hàm cơ bản như root, parent, left, right, sibling, isExternal, và isRoot có thể được thực hiện bằng cách sử dụng các phép toán số học trên các số f(v).

****

****

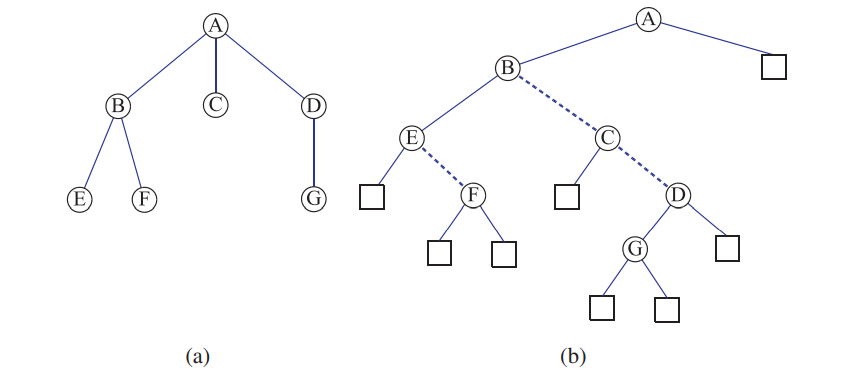
**Bảng 7.3: Thời gian chạy cho một cây nhị phân T được triển khai bằng một vector S. Chúng ta ký hiệu số nút của T là n, và N là kích thước của S. Việc sử dụng không gian là O(N), tương đương với O(2^n) trong trường hợp xấu nhất**

7.3.8 Representing General Trees with Binary Trees.

Một cách biểu diễn thay thế cho cây tổng quát T được đạt được bằng cách chuyển đổi T thành một cây nhị phân T'. (Xem Hình 7.22.) Chúng ta giả định rằng hoặc T được sắp xếp hoặc đã được sắp xếp theo cách tùy ý. Quá trình chuyển đổi như sau:

* Đối với mỗi nút u của T, có một nút nội u' của T' được liên kết với u.
* Nếu u là một nút ngoại của T và không có một nút anh em ngay sau nó, thì các con của u' trong T' là các nút ngoại.
* Nếu u là một nút nội của T và v là con đầu tiên của u trong T, thì v' là con trái của u' trong T'.
* Nếu nút v có một nút anh em w ngay sau nó, thì w' là con phải của v' trong T'.

Lưu ý rằng các nút ngoại của T' không được liên kết với các nút của T và chỉ đóng vai trò như bộ giữ chỗ (do đó, thậm chí có thể là null).

**** Dễ dàng duy trì sự tương ứng giữa T và T', cũng như biểu diễn các thao tác trong T dưới dạng các thao tác tương ứng trong T'. Một cách hiểu intuitively, chúng ta có thể nghĩ về sự tương ứng dưới dạng một quá trình chuyển đổi từ T sang T' mà mỗi tập hợp các anh em {v1, v2, ..., vk} trong T với cha là v được thay thế bằng một chuỗi các con phải có gốc tại v1, sau đó trở thành con trái của v.

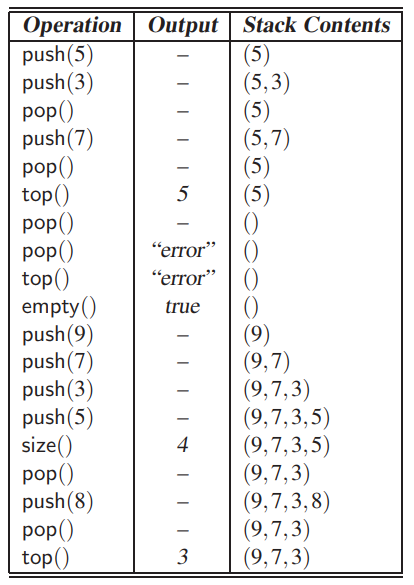
Chapter 5 Stacks, Queues, and Deques

5.1 Ngăn xếp

5.1.1 Abstract Data Type Ngăn Xếp

Ngăn xếp là một trong những cấu trúc dữ liệu đơn giản nhất, nhưng cũng là một trong những quan trọng nhất, vì chúng được sử dụng trong nhiều ứng dụng khác nhau bao gồm nhiều cấu trúc dữ liệu phức tạp hơn. Một cách hình thức, một ngăn xếp là một loại trừu tượng của dữ liệu (ADT) hỗ trợ các thao tác sau:

* push(e): Chèn phần tử e vào đỉnh của ngăn xếp.
* pop(): Loại bỏ phần tử đầu tiên từ ngăn xếp; xảy ra lỗi nếu ngăn xếp là trống.
* top(): Trả về một tham chiếu đến phần tử đầu tiên trên ngăn xếp, mà không loại bỏ nó; xảy ra lỗi nếu ngăn xếp là trống. Ngoài ra, chúng ta cũng định nghĩa các hàm hỗ trợ sau:
* size(): Trả về số lượng phần tử trong ngăn xếp.
* empty(): Trả về true nếu ngăn xếp trống và false nếu ngược lại.
* Example 5.3: Bảng dưới đây thể hiện một loạt các thao tác ngăn xếp và ảnh hưởng của chúng đối với một ngăn xếp số nguyên ban đầu là trống.



5.1.2 Ngăn xếp trong STL

STL cung cấp một ngăn xếp, là một cấu trúc dữ liệu trừu tượng hỗ trợ các thao tác như **push**, **pop**, và **top**. Lớp ngăn xếp của STL được triển khai dựa trên vector STL, có thể được khai báo và sử dụng như sau:

**#include <stack>**

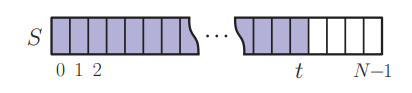
**using std::stack;**

**stack<int> myStack; // Một ngăn xếp số nguyên**

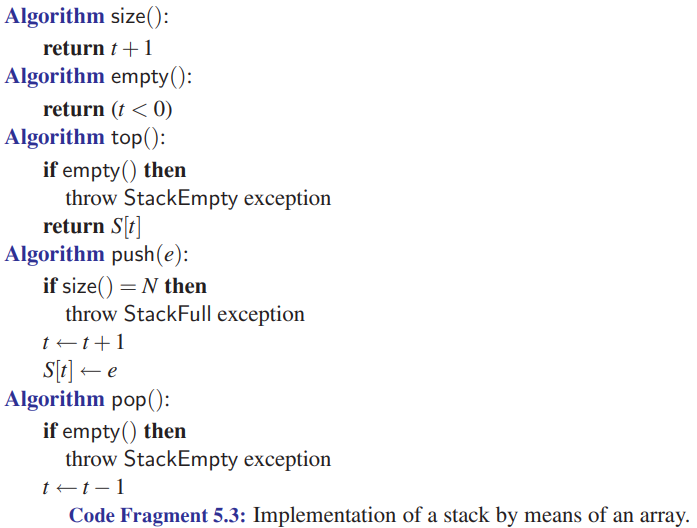
Các thao tác chính bao gồm **size()**, **empty()**, **push(e)**, **pop()**, và **top()**. Một điều quan trọng cần lưu ý là việc áp dụng **top** hoặc **pop** cho một ngăn xếp trống trong STL có kết quả không xác định và không ném ngoại lệ, do đó người lập trình cần đảm bảo tránh truy cập không hợp lệ.

5.1.4 A Simple Array-Based Stack Implementation

Chúng ta có thể triển khai một ngăn xếp bằng cách lưu trữ các phần tử của nó trong một mảng. Cụ thể, ngăn xếp trong cài đặt này bao gồm một mảng S với N phần tử và một biến số nguyên t để chỉ mục của phần tử đầu đỉnh trong mảng S. (Xem Hình 5.2.)

****

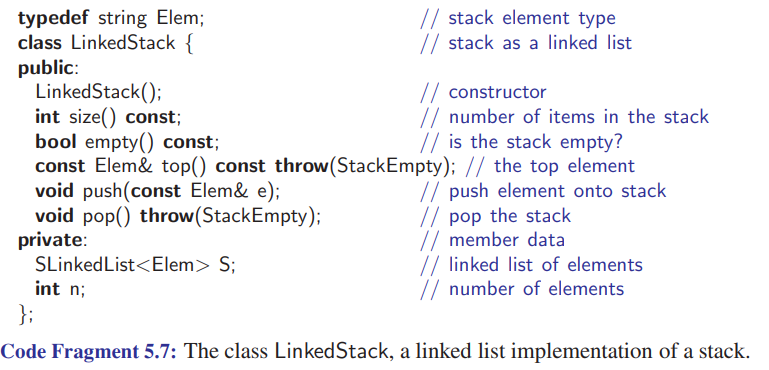
Nhớ rằng mảng trong C++ bắt đầu từ chỉ số 0, chúng ta khởi tạo giá trị của t là -1 và sử dụng giá trị này để xác định khi nào ngăn xếp là rỗng. Tương tự, chúng ta có thể sử dụng biến này để xác định số lượng phần tử trong một ngăn xếp (t + 1). Chúng ta cũng giới thiệu một loại ngoại lệ mới, gọi là **StackFull**, để báo hiệu điều kiện lỗi xuất hiện nếu chúng ta cố gắng chèn một phần tử mới và mảng S đã đầy. Ngoại lệ **StackFull** này chỉ đặc trưng cho cài đặt ngăn xếp của chúng ta và không được định nghĩa trong ADT ngăn xếp. Với ngoại lệ mới này, chúng ta có thể triển khai các chức năng của ADT ngăn xếp như mô tả trong Mã

****

5.1.5 Implementing a Stack with a Generic Linked List

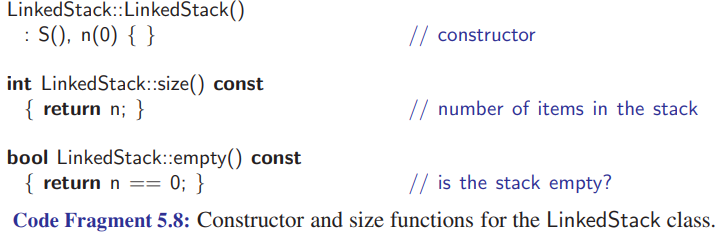
Trong phần này, chúng ta sẽ thấy cách triển khai ADT ngăn xếp bằng cách sử dụng một danh sách liên kết đơn. Phương pháp của chúng tôi là sử dụng danh sách liên kết đơn chung, được gọi là **SLinkedList**, đã được trình bày trước đó trong Phần 3.2.4. Định nghĩa của ngăn xếp của chúng tôi, được gọi là **LinkedStack**, được trình bày trong Đoạn mã 5.7.

Để tránh sự rối bời cú pháp có trong các lớp có temple C++, chúng tôi đã chọn không triển khai một lớp temple toàn diện. Thay vào đó, chúng tôi đã chọn định nghĩa một kiểu cho các phần tử của ngăn xếp, được gọi là **Elem**. Trong ví dụ này, chúng tôi định nghĩa **Elem** là kiểu chuỗi. Chúng tôi để lại nhiệm vụ tạo một cài đặt thực sự chung làm bài tập. (Xem Bài tập R-5.7.)



Thành viên dữ liệu chính của lớp là danh sách liên kết chung kiểu **Elem**, được gọi là **S**. Vì lớp **SLinkedList** không cung cấp hàm thành viên **size**, chúng tôi lưu kích thước hiện tại trong biến thành viên, **n**.

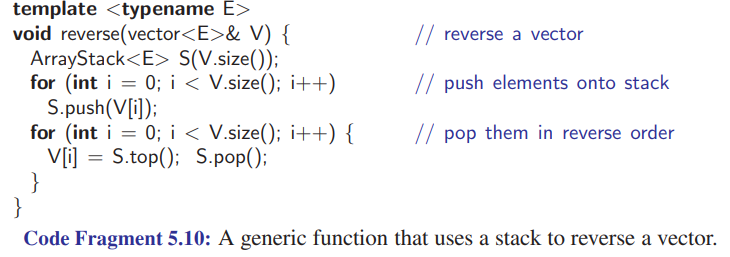
Trong Đoạn mã 5.8, chúng tôi trình bày các cài đặt của hàm tạo và các hàm **size** và **empty**. Hàm tạo của chúng tôi tạo ngăn xếp ban đầu và khởi tạo **n** thành 0. Chúng tôi không cung cấp một hủy bỏ rõ ràng, thay vào đó, chúng tôi phụ thuộc vào hủy bỏ của **SLinkedList** để giải phóng danh sách liên kết **S**.

****

​

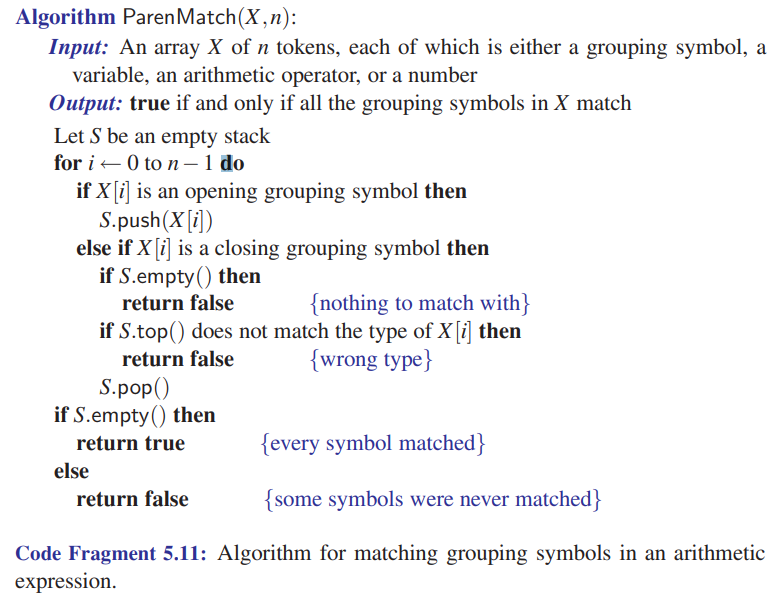
5.1.6 Đảo ngược vectơ bằng cách sử dụng ngăn xếp

Chúng ta có thể sử dụng một ngăn xếp để đảo ngược các phần tử trong một vector, tạo ra một thuật toán không đệ quy cho vấn đề đảo ngược mảng được giới thiệu trong Phần 3.5.1. Ý tưởng cơ bản là đẩy tất cả các phần tử của vector theo thứ tự vào một ngăn xếp và sau đó điền lại vector bằng cách lấy các phần tử ra khỏi ngăn xếp. Ở Đoạn mã 5.10, chúng ta cung cấp một triển khai C++ của thuật toán này.

****

5.1.7 So khớp dấu ngoặc đơn và thẻ HTML

Trong ứng dụng kiểm tra biểu thức toán học, chúng ta sử dụng ngăn xếp để đảm bảo các dấu ngoặc và ký hiệu nhóm khớp đúng. Thuật toán thực hiện quét từ trái sang phải, đẩy dấu mở ngoặc vào ngăn xếp và kiểm tra khi gặp dấu đóng ngoặc. Nếu sau khi xử lý toàn bộ chuỗi mà ngăn xếp trống, thì các ký hiệu khớp đúng. Đây là một thuật toán chạy trong thời gian O(n), với n là chiều dài của chuỗi.



5.2 Queues

5.2.1 Kiểu dữ liệu trừu tượng Hàng đợi

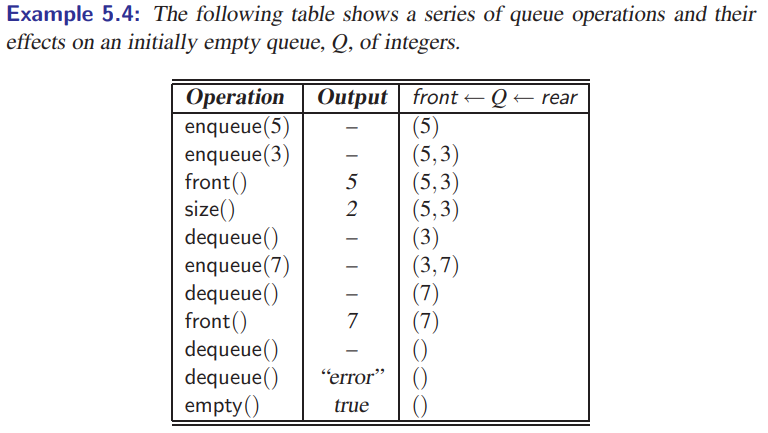
Kiểu dữ liệu trừu tượng Hàng đợi định nghĩa một container giữ các phần tử theo thứ tự, chỉ cho phép truy cập và xóa ở phần đầu (gọi là front) và chèn ở phần cuối (gọi là rear). Điều này tuân theo nguyên tắc đầu tiên vào - đầu tiên ra (FIFO).

Các hoạt động của kiểu dữ liệu trừu tượng Hàng đợi bao gồm:

* enqueue(e): Chèn phần tử e vào cuối hàng đợi.
* dequeue(): Loại bỏ phần tử ở phía trước của hàng đợi; lỗi nếu hàng đợi trống.
* front(): Trả về, nhưng không loại bỏ, tham chiếu đến phần tử ở phía trước; lỗi nếu hàng đợi trống.

Các hàm hỗ trợ bao gồm:

* size(): Trả về số lượng phần tử trong hàng đợi.
* empty(): Trả về true nếu hàng đợi trống và false nếu không.



5.2.2 Hàng đợi trong STL

Thư viện mẫu chuẩn cung cấp một triển khai của hàng đợi. Như làm với ngăn xếp STL, triển khai cơ bản dựa trên lớp vector STL (Mục 1.5.5 và 6.1.4). Để khai báo một đối tượng thuộc kiểu hàng đợi, cần phải bao gồm trước tệp định nghĩa, được gọi là "queue". Tương tự như vector STL, lớp queue là một phần của không gian tên std, nên cần sử dụng "std::queue" hoặc cung cấp một câu lệnh "using" phù hợp. Lớp queue có kiểu mẫu với kiểu cơ sở của các phần tử cá nhân. Ví dụ, đoạn mã dưới đây khai báo một hàng đợi của các số thực:



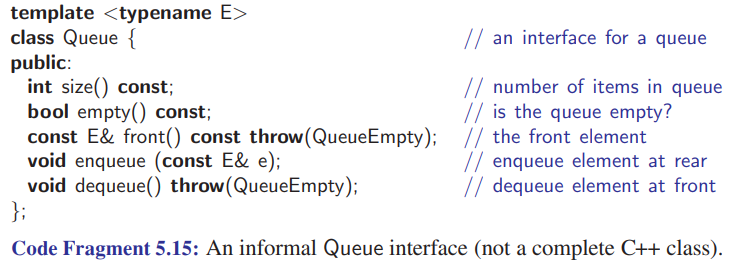
Hàng đợi trong STL tự động thay đổi kích thước khi thêm phần tử mới. Các hàm chính bao gồm:

* size(): Số lượng phần tử trong hàng đợi.
* empty(): Trả về true nếu hàng đợi trống, ngược lại là false.
* push(e): Thêm phần tử e vào cuối hàng đợi.
* pop(): Loại bỏ phần tử ở phía trước của hàng đợi.
* front(): Trả về tham chiếu đến phần tử ở phía trước.
* back(): Trả về tham chiếu đến phần tử ở phía sau.

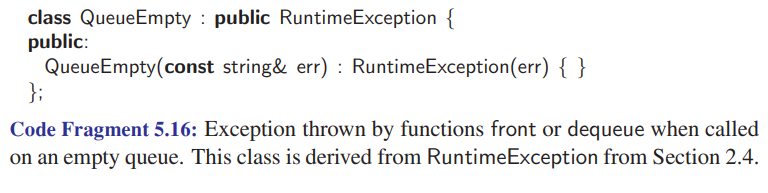
Khác với giao diện của chúng ta, hàng đợi STL cho phép truy cập cả phần đầu và phần đuôi của hàng đợi. Việc thực hiện các hoạt động front, back hoặc pop trên hàng đợi STL trống sẽ có kết quả không xác định và có thể dẫn đến dừng chương trình. Người lập trình cần đảm bảo không có truy cập không hợp lệ nào được thực hiện.

5.2.3 Giao diện Hàng đợi trong C++

Giao diện của chúng tôi cho kiểu dữ liệu trừu tượng Hàng đợi được đưa ra trong đoạn mã 5.15. Tương tự như với kiểu dữ liệu trừu tượng ngăn xếp, lớp này được làm mẫu. Kiểu phần tử cơ bản E của hàng đợi được người dùng cung cấp.



Hàm size và empty có ý nghĩa giống như trong kiểu dữ liệu trừu tượng Ngăn xếp. Các hàm này, cùng với front, được gọi là các hàm truy nhập vì chúng trả về giá trị mà không làm thay đổi cấu trúc dữ liệu. Hàm front trả về tham chiếu hằng số đến đỉnh của hàng đợi. Nếu gọi front hoặc dequeue trên hàng đợi trống, sẽ ném ngoại lệ QueueEmpty để báo lỗi. Các hàm thành viên size, empty, và front được khai báo là const để thông báo cho trình biên dịch rằng chúng không làm thay đổi nội dung của hàng đợi.

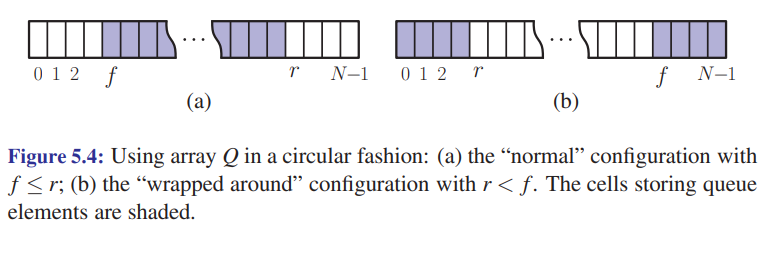


5.2.4 Triển khai đơn giản dựa trên mảng

Chúng tôi triển khai một hàng đợi đơn giản bằng một mảng Q có dung lượng N để lưu trữ phần tử. Chúng tôi sử dụng mô hình mảng hình tròn để giữ theo dõi phần đầu và phần đuôi của hàng đợi mà không cần di chuyển phần tử.

Chúng tôi sử dụng ba biến f (chỉ mục của phần đầu), r (chỉ mục của phần đuôi), và n (số lượng phần tử hiện tại trong hàng đợi). Khi thực hiện dequeue, chúng tôi giảm n và tăng f lên ô tiếp theo. Khi thực hiện enqueue, chúng tôi tăng n và tăng r lên ô tiếp theo. Điều này giúp thực hiện enqueue và dequeue trong thời gian hằng số.

Tuy nhiên, để tránh lỗi khi cố gắng chèn một phần tử khi phần đầu và phần đuôi đều ở chỉ mục N, chúng tôi "vòng qua" chỉ mục cuối cùng của Q. Mảng Q được xem xét như một "mảng hình tròn" từ Q[0] đến Q[N-1] và sau đó quay lại Q[0].



Chúng ta sử dụng toán tử modulo để triển khai một mảng hình tròn. Mỗi khi tăng f hoặc r, chúng ta tính " (f + 1) mod N" hoặc "(r + 1) mod N", nơi "mod" là toán tử modulo. Nếu f và r đạt đến chỉ mục N, chúng ta "vòng qua" cuối của Q để tránh lỗi mảng vượt quá. Triển khai này hiệu quả với thời gian O(1) cho tất cả các chức năng hàng đợi. Tuy nhiên, chúng ta cần đặt dung lượng hàng đợi là một số N, điều này có thể không linh hoạt đối với các ứng dụng thực tế.

5.3 Hàng Đợi Đầu Hai Chiều

Hãy xem xét một cấu trúc dữ liệu giống như hàng đợi nhưng hỗ trợ thêm và xóa ở cả phía trước và phía sau của hàng đợi. Một sự mở rộng như vậy của hàng đợi được gọi là hàng đợi hai đầu, hoặc deque, thường được phát âm là "đêck" để tránh nhầm lẫn với chức năng dequeue của ADT hàng đợi thông thường, được phát âm giống như từ viết tắt "D.Q.". Một cách dễ nhớ cách phát âm "đêck" là quan sát rằng một deque giống như một bộ bài trong tay của một thương nhân bài lá lừa đảo - có thể chia từ cả trên và dưới.

5.3.1 Kiểu Dữ Liệu Trừu Tượng Deque

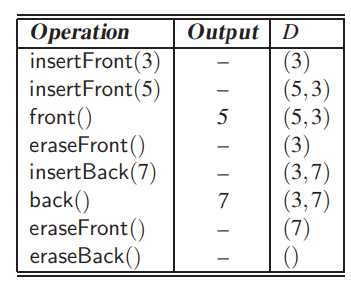
Các chức năng của kiểu dữ liệu trừu tượng Deque là như sau, trong đó D biểu thị cho deque:

* insertFront(e): Chèn một phần tử mới e vào đầu của deque.
* insertBack(e): Chèn một phần tử mới e vào cuối của deque.
* eraseFront(): Loại bỏ phần tử đầu tiên của deque; xảy ra lỗi nếu deque là trống.
* eraseBack(): Loại bỏ phần tử cuối cùng của deque; xảy ra lỗi nếu deque là trống.

Ngoài ra, deque bao gồm các chức năng hỗ trợ sau:

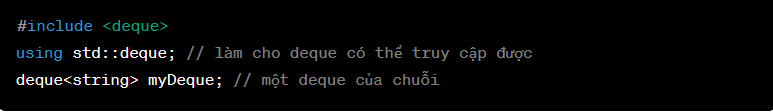
* front(): Trả về phần tử đầu tiên của deque; xảy ra lỗi nếu deque là trống.
* back(): Trả về phần tử cuối cùng của deque; xảy ra lỗi nếu deque là trống.
* size(): Trả về số lượng phần tử của deque.
* empty(): Trả về true nếu deque là trống và false nếu ngược lại.

Ví dụ 5.5: Ví dụ sau đây mô tả một loạt các thao tác và ảnh hưởng của chúng đối với một deque ban đầu trống, D, chứa các số nguyên.



5.3.2 Deque trong STL

Tương tự như stack và queue, Thư viện Mẫu Tiêu Chuẩn cung cấp một phiên bản của deque. Điều triển khai cơ bản dựa trên lớp vector trong STL (Xem các phần 1.5.5 và 6.1.4). Mô hình sử dụng tương tự như stack và queue trong STL. Đầu tiên, chúng ta cần bao gồm tệp định nghĩa "deque". Vì nó là một thành viên của không gian tên std, chúng ta cần hoặc tiền tố mỗi sử dụng "std::deque" hoặc cung cấp một câu lệnh "using" phù hợp. Lớp deque được mẫu hóa với kiểu cơ bản của các phần tử cá nhân. Ví dụ, đoạn mã dưới đây khai báo một deque của chuỗi.



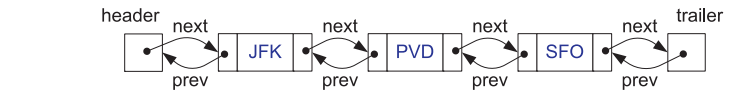
Tương tự như với các stack và queue trong STL, một deque trong STL sẽ tự động điều chỉnh kích thước khi thêm các phần tử mới. Với những sự khác biệt nhỏ, lớp deque trong STL hỗ trợ các toán tử tương tự như giao diện của chúng ta. Dưới đây là danh sách các hoạt động chính.

* size(): Trả về số lượng phần tử trong deque.
* empty(): Trả về true nếu deque là trống và false nếu ngược lại.
* push\_front(e): Chèn e vào đầu deque.
* push\_back(e): Chèn e vào cuối deque.
* pop\_front(): Loại bỏ phần tử đầu tiên của deque.
* pop\_back(): Loại bỏ phần tử cuối cùng của deque.
* front(): Trả về một tham chiếu đến phần tử đầu tiên của deque.
* back(): Trả về một tham chiếu đến phần tử cuối cùng của deque.

Tương tự như với stack và queue trong STL, kết quả của việc áp dụng bất kỳ hoạt động nào như front, back, push\_front hoặc push\_back cho một deque STL trống là không xác định. Do đó, không có ngoại lệ được ném ra, nhưng chương trình có thể bị hủy bỏ.

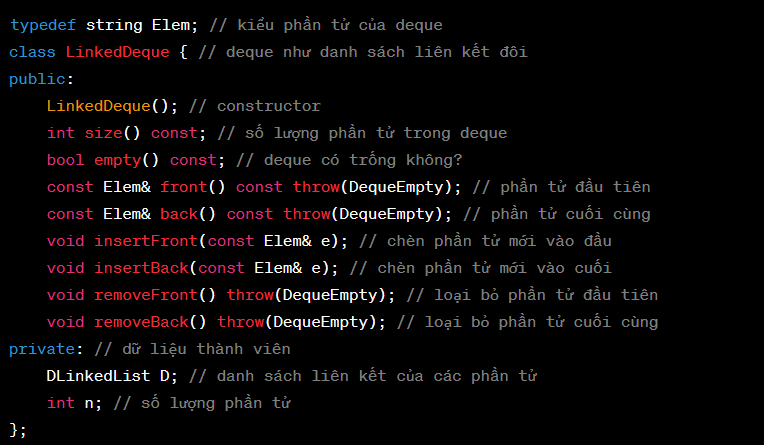
5.3.3 Triển Khai Deque với Danh Sách Liên Kết Đôi

Trong phần này, chúng tôi sẽ chỉ ra cách triển khai kiểu dữ liệu trừu tượng deque bằng cách sử dụng biểu diễn liên kết. Tương tự như với hàng đợi, một deque hỗ trợ việc truy cập hiệu quả ở cả hai đầu của danh sách, do đó, triển khai của chúng tôi dựa trên việc sử dụng danh sách liên kết đôi. Một lần nữa, chúng tôi sử dụng lớp danh sách liên kết đôi, gọi là DLinkedList, đã được trình bày trước đó trong phần 3.3.3. Chúng tôi đặt phía trước của deque tại đầu của danh sách liên kết và phía sau của queue ở cuối. Một minh họa được cung cấp trong Hình 5.7.



Hình 5.7: Một danh sách liên kết đôi với các phần tử gửi, tiêu đề và trailer. Phía trước của deque của chúng tôi được lưu trữ ngay sau tiêu đề ("JFK"), và phía sau của deque được lưu trữ ngay trước trailer ("SFO").

Định nghĩa của lớp kết quả, được gọi là LinkedDeque, được hiển thị trong Đoạn mã 5.21. Deque được lưu trữ trong thành viên dữ liệu D. Để hỗ trợ hàm size, chúng ta cũng duy trì kích thước hàng đợi trong thành viên n. Như trong một số triển khai trước đó, chúng ta tránh sự lộn xộn cú pháp ẩn chứa trong các lớp mẫu C++, và thay vào đó chỉ sử dụng một định nghĩa kiểu để xác định kiểu cơ bản của deque.



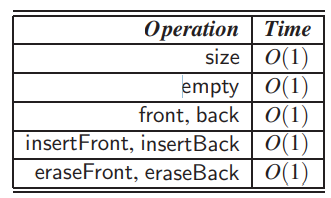
**Đoạn mã 5.21: Cấu trúc lớp cho lớp LinkedDeque**

Chúng tôi không quan tâm đến việc cung cấp một hủy bỏ rõ ràng, vì lớp DLinkedList cung cấp hủy bỏ riêng, được tự động kích hoạt khi cấu trúc LinkedDeque của chúng tôi bị hủy. Hầu hết các hàm thành viên cho lớp LinkedDeque là những tổng quát đơn giản của các hàm tương ứng của lớp LinkedQueue, nên chúng tôi đã bỏ qua chúng. Trong Đoạn mã 5.22, chúng tôi trình bày các triển khai của các hàm thành viên để thực hiện việc chèn và loại bỏ các phần tử từ deque. Chú ý rằng, trong mỗi trường hợp, chúng tôi đơn giản là gọi thực hiện phép toán thích hợp từ đối tượng DLinkedList cơ sở.

Đoạn mã 5.22 trình bày cài đặt các hàm chèn và loại bỏ cho **LinkedDeque**. Bảng 5.2 tóm tắt thời gian chạy của các hàm này trong một biểu diễn của deque sử dụng danh sách liên kết đôi. Lưu ý rằng mọi hàm của kiểu dữ liệu trừu tượng deque trong cài đặt này chạy trong thời gian O(1), cho thấy độ phức tạp thời gian là hằng số cho mỗi thao tác.



**Bảng 5.2 tóm tắt độ phức tạp thời gian của mỗi thao tác trong một deque được triển khai với danh sách liên kết đôi**

****

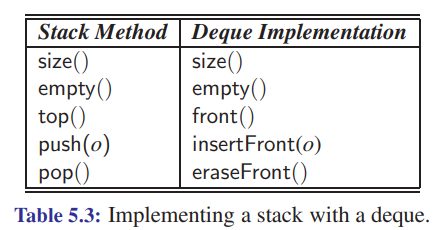
Bảng 5.2 tóm tắt độ phức tạp thời gian của mỗi thao tác trong một deque được triển khai với danh sách liên kết đôi:

5.3.4 Bộ Chuyển Đổi và Mô Hình Thiết Kế Bộ Chuyển Đổi

Kiểm tra các đoạn mã trong Các mục 5.1.5, 5.2.5 và 5.3.3, ta nhận thấy một mẫu chung. Trong mỗi trường hợp, chúng ta đã lấy một cấu trúc dữ liệu hiện tại và điều chỉnh nó để được sử dụng cho một mục đích đặc biệt. Ví dụ, ở Mục 5.3.3, chúng ta đã chỉ ra cách lớp **DLinkedList** từ Mục 3.3.3 có thể được điều chỉnh để triển khai một deque. Ngoại trừ tính năng bổ sung theo dõi số lượng phần tử, chúng ta đã đơn giản là ánh xạ mỗi hoạt động deque (như **insertFront**) thành hoạt động tương ứng của **DLinkedList** (như **addFront**).

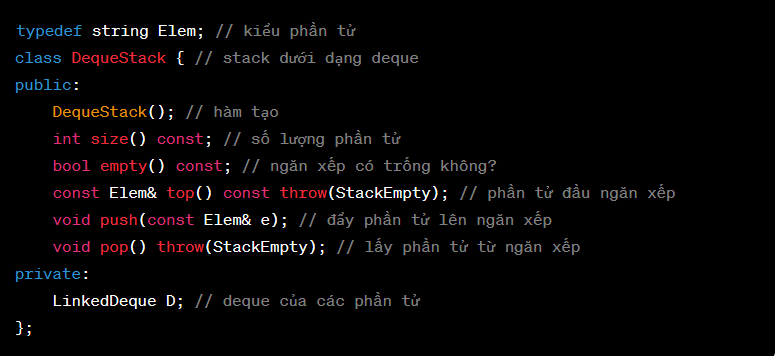
Một bộ chuyển đổi (đôi khi được gọi là bọc) là một cấu trúc dữ liệu, ví dụ như một lớp trong C++, chuyển đổi một giao diện thành giao diện khác. Bạn có thể nghĩ về một bộ chuyển đổi như là phương tiện phần mềm tương đương với bộ chuyển đổi đầu cắm điện, thường cần khi bạn muốn cắm các thiết bị điện của mình vào ổ cắm điện ở các quốc gia khác nhau.

Là một ví dụ về việc điều chỉnh, hãy nhận thức rằng có thể triển khai kiểu dữ liệu trừu tượng stack bằng cách sử dụng một cấu trúc dữ liệu deque. Nói cách khác, mỗi hoạt động của stack có thể được chuyển đổi thành một hoạt động deque có chức năng tương đương. Một ánh xạ như vậy được trình bày trong Bảng 5.3.



Là một ví dụ cụ thể hơn về mô hình thiết kế bộ chuyển đổi, hãy xem xét đoạn mã được hiển thị trong Đoạn mã 5.23. Trong đoạn mã này, chúng ta trình bày một lớp **DequeStack**, thực hiện kiểu dữ liệu trừu tượng stack. Việc triển khai của nó dựa trên việc chuyển đổi mỗi hoạt động của stack thành hoạt động tương ứng trên một **LinkedDeque**, đã được giới thiệu trong Mục 5.3.3.

**Đoạn mã 5.23: Triển khai giao diện Stack bằng cách sử dụng một deque.**

****

**Đoạn mã 5.24: Triển khai giao diện Stack bằng cách sử dụng một deque.**

****