Cây tiền tố

Bài tập môn: Cấu trúc Dữ liệu và Thuật toán

Trường Đại học Khoa học Tự nhiên, ĐHQG Hà Nội

Lê Hồng Phương, <phuonglh@gmail.com>, 2015.

1. Cây tiền tố

Cây tiền tố là một cấu trúc dữ liệu được sử dụng rộng rãi trong tin học, thường dùng để biểu diễn một cách hiệu quả một từ điển gồm nhiều từ khác nhau.

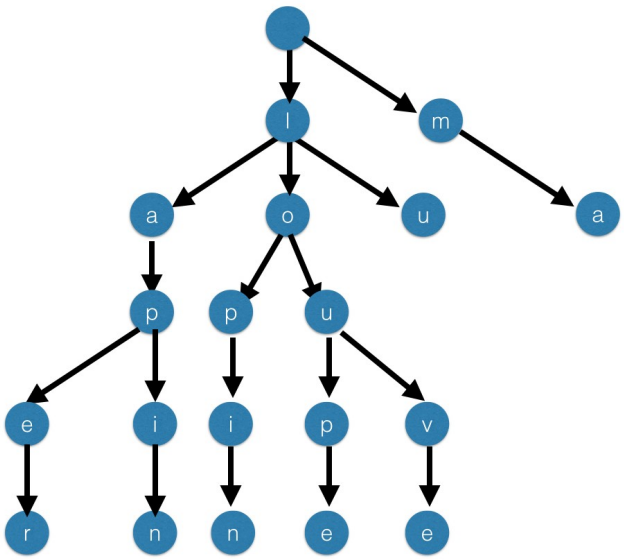
1.1. Nguyên lí của cây tiền tố

Trong bài tập này, ta sẽ mã hóa một từ điển, là một cấu trúc dữ liệu tốt để chứa một tập từ. Từ điển cho phép ta một chuỗi kí tự có phải là tiền tố của một từ trong từ điển này không. Ví dụ, chuỗi "dict" là một tiền tố của từ "dictionary". Trong nhiều ứng dụng, ta cần thường xuyên tìm kiếm các chuỗi kí tự nên việc biểu diễn và xử lí hiệu quả từ điển là một vấn đề quan trọng. Đây là vấn đề tổng quát, không phụ thuộc vào ngôn ngữ tự nhiên nào, ví dụ tiếng Anh, tiếng Pháp, hay tiếng Việt.

Xét ví dụ gồm 7 từ sau:

lapin, laper, lopin, louve, loup, loupe, ma.

Ta cần xây dựng một cấu trúc dữ liệu cho phép lưu các từ này và trả lời một số dạng câu hỏi sau: Từ **loup** có nằm trong tập từ này không? Có từ nào bắt đầu bằng **lap** không và nếu có thì có những từ nào có thể là từ tiếp theo?



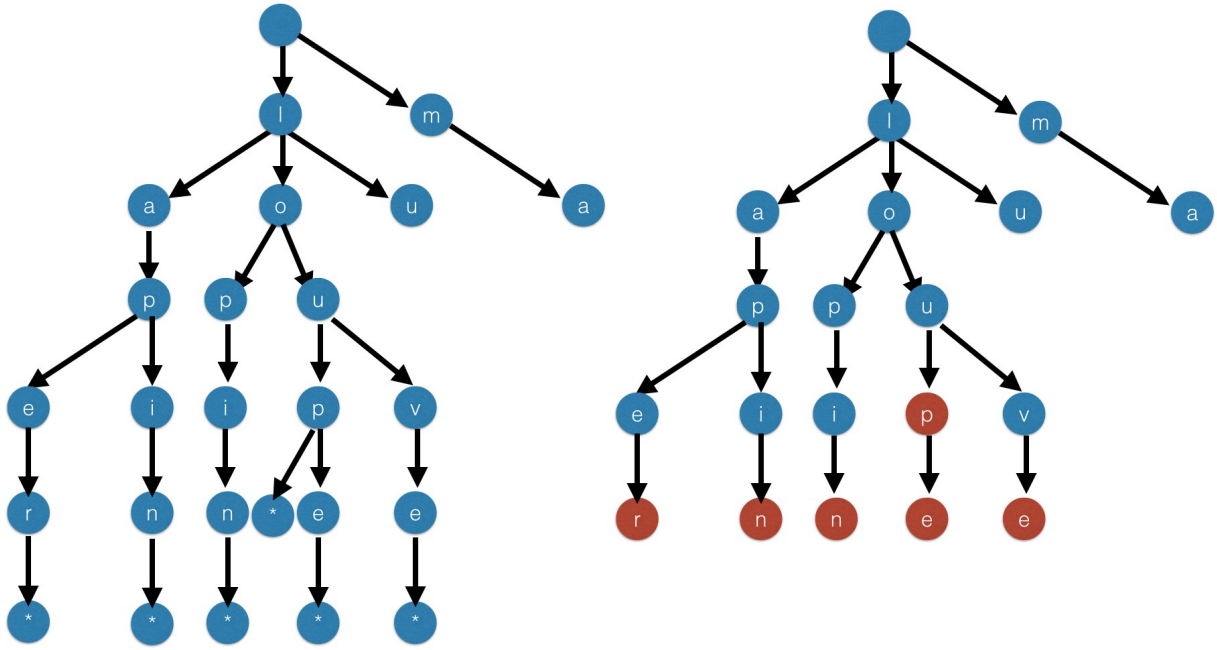
Để thực hiện những thao tác này, ta sẽ sử dụng phương pháp mã hóa dữ liệu bằng cây *n*-phân.

Tất nhiên, ta cũng có thể sử dụng một mảng được sắp xếp để lưu các từ dưới dạng các xâu kí tự. Nhưng cách lưu trữ này có hai nhược điểm. Một là cấu trúc dữ liệu này tốn nhiều bộ nhớ. Hai là, thao tác thêm một từ vào từ điển sẽ cần độ phức tạp thời gian tuyến tính theo kích thước của từ điển.

Nguyên lí của cây n-phân khá đơn giản. Trong ví dụ trên, cây n-phân được sử dụng sẽ có dạng như hình trên.

1.2. Đánh dấu hết từ

Phương pháp mã hóa bằng cây n-phân như trên có nhược điểm là khó phân biệt các từ với nhau. Ví dụ, làm sao biết rằng chuỗi **loup** là một từ trong từ điển? Để khắc phục nhược điểm này, ta sẽ dùng một kí tự đặc biệt là kí tự * để đánh dấu hết từ. Như vậy, cây sẽ có hai kiểu nút: một kiểu nút để biểu diễn các chữ cái của mỗi từ và một kiểu nút biểu diễn kí hiệu hết từ. Ta biểu diễn cây n-phân như hình dưới đây.



1.3. Viết lớp Node

Các lớp **Node** và **Dictionary** được viết trong tệp *Dictionary.java*. Hãy viết lớp **Node**. Chú ý rằng mỗi nút chứa một kí tự. Mặc định thì mỗi nút được tạo ra chưa có nút con. Tiếp theo, viết hàm **addChild(Node a)** để thêm một nút con cho nút hiện tại. Dưới đây là ví dụ minh họa cách sử dụng lớp này:

```
Node a1 = new Node('*');
Node a2 = new Node('*');
Node a3 = new Node('*');
Node b = new Node('l');
```

```
Node c = new Node('e');
Node d = new Node('a');
Node e = new Node('s');
b.addChild(c);
b.addChild(d);
c.addChild(e);
c.addChild(a1);
d.addChild(a2);
e.addChild(a3);
```

Ta thấy rằng các lệnh trên tạo ra cây ứng với tập từ "**le**", "**la**", "**les**".

Cấu trúc dữ liệu nào là thích hợp để biểu diễn một danh sách các nút con của một nút?

Để kiểm tra việc cài đặt các lớp, ta viết thêm hàm hiển thị thông tin về mỗi nút. Hãy viết hàm **public String toString()** để in ra một nút dưới định dạng xâu, theo cách như sau. Trong ví dụ trên, khi in nút b bằng lệnh

```
System.out.println(b);
```

ta sẽ được kết quả

```
l(e(s(*), *), a(*))
```

2. Mã hóa từ điển

Sau khi viết xong lớp **Node**, bạn viết lớp **Dictionary**. Lớp này biểu diễn một từ điển gồm một tập từ khác nhau.

Trong các hình vẽ ở trên, ta dùng một nút gốc, là nút không chứa kí tự nào, nhằm làm cho việc duyệt cây được dễ dàng. Ta có thể dùng một kí tự đặc biệt, chẳng hạn dấu gạch dưới _ để biểu diễn kí tự của nút gốc. Từ điển rỗng tương ứng với một cây chỉ có một nút gốc. Hãy viết lớp **Dictionary** và hàm tạo của nó. Lớp này lưu tham chiếu tới nút gốc của cây.

2.1. Kiểm tra từ

Tiếp theo, viết hàm kiểm tra sự tồn tại của một từ **m** trong từ điển. Để làm việc này, ta lần lượt đọc từng kí tự của **m**, xuất phát từ nút gốc của từ điển, kiểm tra xem kí tự tiếp theo có nằm trong một trong các nút con của nút hiện tại không. Ví dụ, nếu **m** là từ **lapa**, từ gốc ta đi tới nút con chứa kí tự **l**. Tiếp theo ta đi tới nút con của **l** chứa kí tự **a**, và đến tiếp nút chứa kí tự **p**. Tại đây, ta thấy không có nút con nào chứa kí tự **a**, nên ta kết luận là từ **lapa** không có trong từ điển. Tuy nhiên, chú ý rằng ngay cả khi ta đã đọc hết các kí tự của chuỗi **m**, thì chuỗi này chưa chắc đã là một từ. Ví dụ, ta có thể đọc hết chuỗi **lo** nhưng chuỗi này không phải một từ. Như vậy, khi hết từ, ta cần kiểm tra xem nút hiện tại có phải là nút * hay không mới kết luận rằng chuỗi là một từ trong từ

điển.

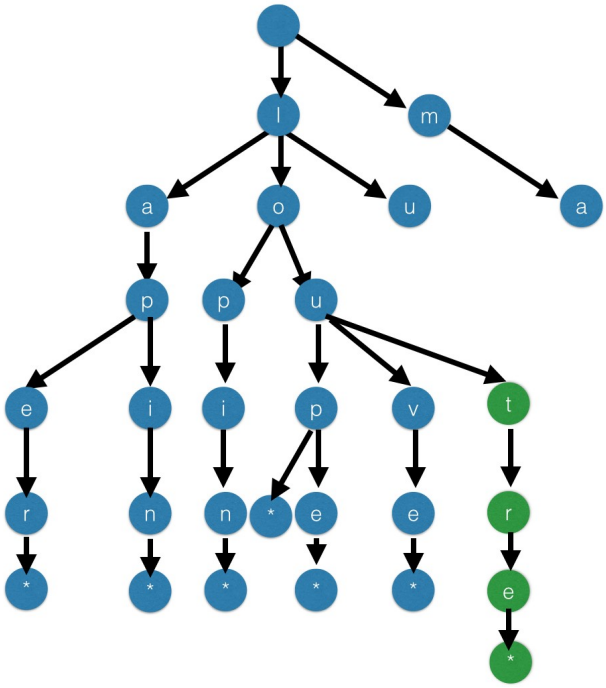
Viết hàm **public boolean existWord(String s)** trong lớp **Dictionary**.

Ta có thể sử dụng cách sau để giúp hoàn thành hàm này dễ dàng hơn. Viết một hàm đệ quy trong lớp Node có dạng sau **public boolean existWordRecursive(String s, int pos)** để kiểm tra xem nút hiện tại có một con chứa kí tự **s.charAt(pos)** hay không. Nếu có thì ta đọc phần tiếp theo của xâu **s**, nếu không thì có thể kết luận là xâu **s** không phải là một từ trong từ điển. Chú ý xử lí trường hợp cuối khi **pos** bằng độ dài của xâu.

Chạy hàm **test1()** và kiểm tra kết quả.

2.2. Xây dựng từ điển

Bây giờ, ta xây dựng từ điển từ một danh sách các từ. Viết hàm **public boolean addWord(String s)** trong lớp **Dictionary,** hoạt động như sau: xuất phát từ đầu xâu và nút gốc, mỗi khi đọc một kí tự thì ta di chuyển tới nút con tương ứng; khi không tìm thấy nút con nào ứng với kí tự hiện tại thì bắt đầu tạo mới một nhánh cây con tương ứng với các kí tự còn lại. Ví dụ, giả sử ta cần thêm từ **loutre** vào từ điển hiện tại. Xuất phát từ đầu từ và từ nút gốc, ta đi theo các nút **gốc** → **l** → **o** → **u**. Tới đây, ta tạo một nhánh mới gồm các nút tương ứng với các kí tự còn lại "**tre**" của xâu **s**.



Lập trình hàm **addWord(String s)**. Hàm này trả về **true** nếu từ **s** được thêm vào từ điển, trả về **false** nếu **s** là từ đã có trong từ điển.

Chạy hàm **test2()** và kiểm tra kết quả.

2.3. Kiểm tra tiền tố

Tiếp theo, viết hàm **public isPrefix(String s)** kiểm tra xem một xâu s có phải là tiền tố của một từ trong từ điển hay không.

Chạy hàm **test3()** và kiểm tra kết quả.

2.3. Liệt kê các từ trong từ điển

Bây giờ, ta thực hiện liệt kê các từ trong từ điển theo thứ tự. Việc này được quy về việc duyệt cây.

Viết hàm **public void listWords()** để liệt kê mọi từ của từ điển theo thứ tự. Các từ cần cách nhau bằng một kí tự trắng.

Có một số khó khăn kĩ thuật sau.

Thứ tự từ điển

Khó khăn đầu tiên là các từ cần được liệt kê theo thứ tự. Danh sách các nút có thể chưa được sắp. Trước tiên, ta cần sửa lại mã nguồn sao cho các danh sách các nút luôn được sắp theo thứ tự.

Hãy xem lại hàm **addWord(String)** đã viết**.** Cách tiếp cận đơn giản ở trước có thể dùng cấu trúc **LinkedList<Node>** để chứa danh sách các nút con của mỗi nút, và mỗi khi thêm một nút con thì chỉ cần gọi hàm **add(Node)** của **LinkedList** để thêm nút đó vào cuối danh sách. Bây giờ, mỗi khi thêm một nút, bạn cần thêm nút đó vào đúng vị trí của nó thay vì thêm vào cuối. Để làm điều này, bạn có thể sử dụng hàm **add(int index, Node n)** của **LinkedList** để thêm một nút vào vị trí mong muốn.

Khi viết hàm này, bạn cần lưu ý:

- thứ tự cần hiển thị là thứ tự từ điển, chẳng hạn từ “bout” đi trước từ “bouteille”;
- các trường hợp giới hạn: có thể cần chèn nút vào đầu hoặc cuối danh sách;
- kiểm tra bằng tay với một số danh sách có độ dài 0, 1 hoặc 2.

Hàm hiển thị các nút đã viết ở trước sẽ giúp ích trong việc đảm bảo rằng mã nguồn bạn viết là đúng.

Ngăn xếp kí tự

Khó khăn thứ hai là lưu danh sách các kí tự đã duyệt để mỗi khi ta gặp kí tự '*' thì ta có thể hiển thị luôn từ đó. Có nhiều cách làm.

Có một cách đơn giản là lưu một danh sách móc nối các kí tự đã duyệt. Viết hàm **public void listWords(LinkedList<Character> prefix)** trong lớp **Node**. Hàm này lưu danh sách **prefix** chứa các kí tự đã gặp trên đường đi tới nút hiện tại. Lớp **Dictionary** sẽ gọi hàm này sau khi truyền vào một danh sách rỗng. Trước khi thực hiện gọi hàm đệ quy để di chuyển xuống các nút con thì hàm này cần mở rộng danh sách bằng việc thêm vào danh sách kí tự hiện tại.

Hàm này cũng cần thêm một công việc phụ như sau. Bạn cần viết một hàm để chuyển một **LinkedList<Character>** thành **String**, sử dụng lớp **StringBuilder**.

Chú ý, để thực hiện tác vụ này, bạn cần kết hợp kí thuật đệ quy với kĩ thuật quay lui để khôi phục lại trạng thái của các danh sách **prefix**.

Chạy hàm **test4()** và kiểm tra kết quả.