Trie (Cây tiền tố)

18127080 , 18127221 , 18127231

December 2019

1 Giới thiệu

1.1 Thông tin nhóm

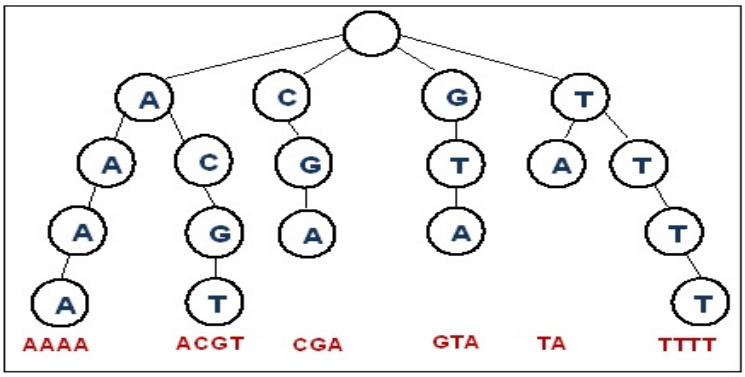
Tên : **Bùi Văn Thiện**
Mã số sinh viên : **18127221**

Tên : **Đoàn Đình Toàn**
Mã số sinh viên : **18127231**

Tên : **Kiều Vũ Minh Đức**
Mã số sinh viên : **18127080**

2 Nội dung tìm hiểu phần lý thuyết

2.1 Độ phức tạp của các thao tác trên Tries



Hình 1: Ví dụ

2.1.1 Thao tác thêm một từ (Insert)

```
put(p, key, val, d) {
   if (p == NULL) {
      p = new node;
      p->value = NIL;
      for (int i = 0; i < |Σ|; i++)       p->next[i] = NULL;
   }
   if (d == strlen(key)) {
      p->value = val;
      return p;
   }
   c = key[d];
   p->next[c] = put(p->next[c], key, val, d + 1);
   return p;
}
Insertion(root, key, val) {
   root = put(root, key, val, 0);
}
```

Hình 2: Thêm một từ vào Trie

Đối với thao tác thêm một từ vào Trie,hàm **put()** được gọi đệ quy với số lần bằng độ dài của *string* truyền vào.

Giả sử cần thêm từ **GTA** vào một Trie không phải *NULL*, lần lượt thêm vào **node G** và tăng $\mathbf{d = 1}$, sau đó thực hiện việc đó đến khi **d** bằng với độ

dài của *string*, và gán cho biến **val** một tín hiệu để biết rằng ở **node** đó có thể đã kết thúc một từ. Do đó độ phức tạp của việc thêm vào một từ có độ dài là **s** sẽ có độ phức tạp là $O(s)$, nếu Trie có **n** từ, thì độ phức tạp của cả Trie là $O(s * n)$

2.1.2 Thao tác xóa một từ (Delete)

```
del(p, key, d) {
   if (p == NULL)             return NULL;
   if (d == strlen(key))
      p->value = NIL;
   else {
      c = key[d];
      p->next[c] = del(p->next[c], key, d + 1);
   }
   if (p->value != NIL)             return p;
   for (c = 0; c < |Σ|; c++)
      if (p->next[c] != NULL)       return p;
   delete   p;
   return NULL;
}
Deletion(root, key) {
   root = del(root, key, 0);
}
```

Hình 3: Xóa một từ khỏi Trie

Sử dụng lại ví dụ trên, giả sử xóa đi từ **GTA**, ta sẽ gọi đệ quy để duyệt **node G**, sau đó duyệt tiếp children của **G** và đệ quy đến hết độ dài của **key**, nếu thỏa điều kiện cần là **node** có tồn tại, chúng ta sẽ xem từ đó đã được định nghĩa chưa bằng việc kiểm tra tín hiệu **value** có khác *NULL* hay không, nếu bằng *NULL* thì tức là **key** này chưa được định nghĩa, và chúng ta không cần làm gì.

Ngoài ra xét tiếp tục trường hợp xem **key** đang xét có phải là một **substring** của một key khác không. Nếu **key** đang xét là một **substring** thì chúng ta cũng không cần làm gì, chỉ việc set lại **value** là *NULL*, còn không thì thực hiện việc xóa từng **node** lá có trong **key** đó.

Nhìn chung thao tác này cũng là duyệt qua từng phần tử của **key** bằng đệ quy nên số lần gọi đệ quy bằng với độ dài của **key**, do đó độ phức tạp của thao tác tìm kiếm một **key** có độ dài **s** là $O(s)$

2.1.3 Thao tác tìm kiếm một từ (Find)

```
get(p, key, d) {
   if (p == NULL)             return NULL;
   if (d == strlen(key))      return p;
   c = key[d];
   return get(p->next[c], key, d + 1);
}
findNode(p, key) {
   p = get(p, key, 0);
   return   (p && p->value != NIL) ? p : NULL;
}
```

Hình 4: Tìm một từ trên Trie

Đối với thao tác tìm kiếm một từ trên Trie, hướng xử lý cũng là việc gọi đệ quy để duyệt qua các phần tử, nếu tồn tại các phẩn tử của **key** trong Trie, chúng ta xem xét từ đó đã được định nghĩa chưa, bằng việc kiểm tra tín hiệu **value** có khác *NULL*, nếu khác *NULL* thì **key** đó đã được định nghĩa trong Trie. Thông qua các bước trên, độ phức tạp của thao tác tìm kiếm phụ thuộc vào độ dài của **key**, do đó độ phức tạp của thao tác tìm kiếm một **key** có độ dài **s** là $O(s)$

2.1.4 Thao tác tìm tất cả các từ có cùng tiền tố i (Find)

```
collect(p, prefix, d) {
   if (p == NULL)       return;
   if (p->value != NIL)
      cout << p->value << " " << prefix << endl;
   for (c = 0; c < |Σ|; c++)
      collect(p->next[c], <prefix + c>, d + 1);
}
get(p, key, d) {
   if (p == NULL)             return NULL;
   if (d == strlen(key))      return p;
   c = key[d];
   return get(p->next[c], key, d + 1);
}
keysWithPrefix(root, prefix) {
   ref p = get(root, prefix, 0);
   d = strlen(prefix);
   collect(p, prefix, d);
}
```

Hình 5: Tìm các từ có cùng tiền tố i

Đối với thao tác này, trường hợp best case là **i** bằng độ dài của từ dài đã được định nghĩa trong Trie, lúc này thì kết quả trả về có duy nhất một từ, do đó nếu **i** là từ dài nhất thì độ phức tạp là $O(i)$

Với trường hợp worst case, là **i** không nhập gì, thì lúc này cần phải duyệt toàn bộ cây để lấy kết quả, do đó giả sử cây có **h** tầng (**h** = độ dài của từ dài nhất có trong dict), **k** là số child trung bình của một **Node**, thì độ phức tạp là $O(k^h)$

Còn với trường hợp trung bình, cây sẽ duyệt tiền tố **i**, mất chi phí là $O(i)$, sau đó duyệt toàn bộ phần còn lại mất chi phí là $O(k^{h-i})$. Do đó chi phí cho việc tìm tất cả từ có cùng tiền tố **i** có độ phức tạp là $O(k^{h-i} - i)$

2.1.5 So sánh với các cấu trúc dữ liệu khác

Nhìn chung các thao tác xử lý trên Trie có chi phí là $O(s)$ với **s** là độ dài của index thao tác đó.

- So sánh với Hash table :

  Tuy phần lớn các thao tác thực hiện trên hash table là $O(1)$ nếu được cài đặt hợp lý, tuy nhiên hash table lại không thích hợp cho thao tác tìm các từ có cùng thành phần (tính năng auto-complete) do hash table

không có khả năng biểu diễn mối tương quan giữa 2 từ gần giống nhau. Nếu cài đặt tính năng auto-complete cho hash table, điều này sẽ mất chi phí là $O(n)$ do cần phải duyệt toàn bộ table.

Nếu phải xử lý collision trên hash table, worst case của thao tác duyệt là $O(n)$, nhưng đối với Trie, worst case cho thao tác này chỉ là $O(s)$ với **s** là độ dài của từ khóa cần duyệt.

Ngoài ra việc quản lý vùng nhớ của Trie cũng tối ưu hơn hash table, trong khi Trie dùng đến đâu cấp vùng nhớ đến đó, còn hash table thì cần phải tạo mảng trước khi thao tác, dẫn đến vùng nhớ bị dư thừa lớn nếu cài đặt không hợp lý.

- So sánh với Binary Search Tree (BST) :

  Do Trie cũng là một cấu trúc Tree, nên Trie có một vài điểm tốt hơn.

  Đối với thao tác tìm kiếm, ở BST cần phải mất $O(log(n))$, trong khi ở Trie, để tìm kiếm một từ, độ phức tạp chỉ là $O(s)$ với **s** là độ dài của từ khóa cần duyệt.

  Do BST bị bó buộc bởi việc một **node** chỉ được có 2 child, nên việc tổ chức cũng gặp nhiều khó khăn, đặc biệt đối với kiểu dữ liệu là *string* hoàn toàn tối ưu hơn cho Trie hơn là BST.

2.2 Ý tưởng giải quyết bài toán ở phần lập trình

Với bài toán được đưa ra, nhóm em có hai hướng tiếp cận để giải quyết. Sau đó có thực hiện việc so sánh với nhau để đưa ra lựa chọn tốt nhất.

- Hướng thứ nhất : Sử dụng HashTable

  Ở phương pháp tiếp cận này, nhóm em đã nghĩ ra một hướng giải quyết, đó là gán các chữ cái là số nguyên tố, mục đích của việc này là khi hash, phép tính đó có thể truy ngược lại ra được tạo thành từ ký tự nào.

  Ví dụ như cho *string* là **"a b"**, qua việc map thì dễ dàng định nghĩa ký tự $\mathbf{a = 2\ b = 3}$, thì qua việc hash là nhân các số đã được map sẵn, kết quả sẽ trả ra là 6. Giả sử lúc duyệt ta có số 6 thì ta có thể truy ngược lại là được tạo ra từ phép $2 * 3$ hoặc $3 * 2$, từ đó có thể xác định được tạo từ 2 ký tự là **a** và **b**.

  Các thao tác như Hash, tìm theo yêu cầu, nhìn chung sẽ có chi phí là $O(1)$, tuy nhiên chi phí cho việc tạo query cho **n** ký tự input sẽ mất chi phí là $O(2^n)$, giả sử nhập vào **a b**, thì kết quả trả ra sẽ là **a , b , ab**

, ba, sau khi đã liệt kê các trường hợp có thể, sẽ đối chiếu lại với file **dict.txt**. Nên độ phức tạp của toàn bộ thao tác tìm kiếm bằng hướng này sẽ có chi phí là $O(2^n)$

- Hướng thứ hai : Sử dụng Trie

  Đối với hướng tiếp cận này, nhóm em cài đặt cây tiền tố để giải quyết bài toán. Đối với thao tác tìm kiếm theo yêu cầu thì độ phức tạp là $O(k! * n)$ với **k** là số ký tự phân biệt và **n** là số ký tự của input. Giả sử input là **"a a b"** thì độ phức tạp của thao tác này là $O(2! * 3) = 6$ phép toán.

2.2.1 So sánh các phương pháp giải quyết

Giữa hai phương pháp này có một số điểm mạnh riêng, tùy thuộc vào data trong file **dict.txt** và input đầu vào để xử lý.

Do phần lớn các thao tác dựa trên HashTable khi thực hiện yêu cầu của đồ án là $O(1)$, và chi phí đã phân tích như trên là $O(2^n)$ với **n** là số lượng ký tự input. Ở đây độ phức tạp tương đối ổn định, ít bị ảnh hưởng từ input và file **dict.txt**.

Tuy nhiên trong thực tế, việc các từ ngữ có nhiều chữ cái trùng nhau gây khó khăn cho HashTable. Giả sử có query sau :

a a a a b b b c c c c d d d d e e e e f f f f g g g g h h h h i i i i j j k k k k l l l l m m m m n n n n o o o o o p p p q q r r r r s s s s s t t t t u u u u v v v w w w w x x x y y y z z z z

(query này đã được nhóm em chạy thực nghiệm và in ra). Trường hợp này thì HashTable không thể xử lý được, hiệu số giữa số lượng input $\mathbf{n} = 97$ và số lượng ký tự phân biệt $\mathbf{k} = 26$ là quá nhiều, do đó ở trường hợp này thì Trie xử lý tốt hơn HashTable, do phân tích ở trên, chi phí của HashTable cho trường hợp này là 2^{97} còn của Trie chỉ là $26! * 97$.

3 Tham khảo

Nhóm em có tham khảo một số nguồn để hoàn thành đồ án :

- `http://www.cplusplus.com/reference/locale/numpunct/grouping/`
- `https://www.geeksforgeeks.org/trie-insert-and-search/`
- `https://www.techiedelight.com/remove-duplicates-vector-cpp/`
- Ngoài ra cũng có sự hướng dẫn của các giảng viên trợ giảng và của tài liệu từ Big-O Coding