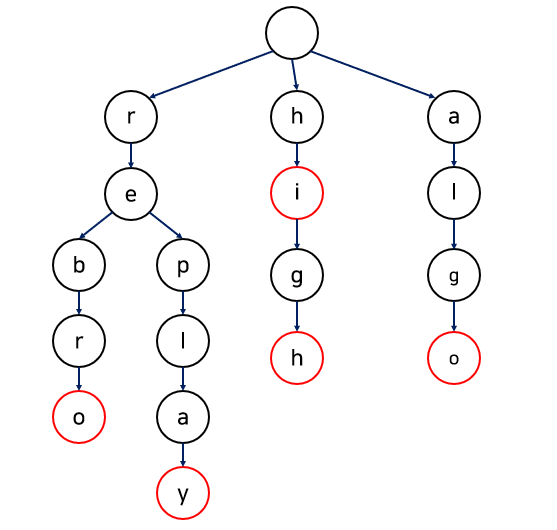
Trie

문자열들을 저장하기 위해 사용되는 트리 자료구조

각 노드가 26개의 자식 포인터를 가짐

빠른 문자열 탐색 속도

작동 원리



문자열들이 저장되어 있는 Trie 의 모습이다.

저장된 문자열 : { rebro , replay , hi , high , algo }

문자열 한 글자씩 노드에 저장 해두었고 끝나는 지점을 bool finish 변수를 통해 표시 해두었다. ( 빨간색 동그라미 부분 -> finish 변수 true)

문자열 탐색하는 과정을 알아보자

문자열 High를 찾는 다고 가정을 해보겠다.

제일 꼭대기에 있는 Root 노드 에서 High의 첫 글자인 ‘H’를 먼저 찾는다.

자식 노드 들 중 H 노드가 존재하기에 H 노드로 이동 할 수 있다.

이동한 뒤 두 번째 글자인 I를 찾는다.

I도 있고 그 다음 G , H도 존재 하기 때문에 H까지 이동할 수 있다.

H가 마지막 글자이기 때문에 H 노드에서 finish 변수가 true 인지 체크 한다.

H 노드의 finish가 true( 빨간색 동그라미 )이므로 저장된 문자열이 맞다. ( 탐색 완료 )

요약

Root -> ‘H’ 노드 이동 -> ‘I’노드 이동 -> ‘G’노드 이동 -> ‘H’노드 이동 -> finish 체크 ( True , 빨간색 동그라미 ) -> 탐색 성공 ! 문자열 존재

존재 하지 않는 문자열들 Case 2 가지

Case 1) 노드 존재 X

위의 Trie에서 “”HILL” 문자열을 찾는다고 가정해보자

Root -> ‘H’ 노드 이동 -> ‘I’ 노드 이동 -> ‘L’ 노드 존재 X -> 탐색 실패!

Case 2) 노드 존재 , finish가 false

위의 Trie에서 “HIG” 문자열을 찾는다고 가정해보자

Root -> ‘H’ 노드 이동 -> ‘I’ 노드 이동 -> ‘G’ 노드 이동 -> finish 체크 ( False, 검정색 동그라미 ) -> 탐색 실패 !

만약 문자열 “HIG”을 Trie에 저장되어 있는 지 확인 해본다고 해보자.

장점 / 단점

문자열 탐색이 굉장히 빠르다(색깔 다르게)

* 탐색의 속도가 O( log n ) 의 시간 복잡도를 가진 ‘이진 탐색 트리’ ( Binary Search Tree ) 를 통해 문자열을 탐색하면 O ( 문자열의 길이 \* log n ) 만큼의 시간복잡도가 걸림
* Trie의 경우 모든 문자열을 O ( log n ) 로 탐색이 가능 하다.

하지만 메모리를 많이 사용하는 단점이.. (색깔 다르게)

알파벳은 총 26개이므로 , 문자열을 표현하는 각 노드가 26개의 자식 노드 포인터를 가진다.

포인터의 크기는 32 비트 / 64 비트 기준 각각 4bytes / 8bytes 이다.

64 비트 기준으로 포인터의 메모리는

8 bytes \* 26개 \* 노드의 갯수 ( 문자열의 길이 \* 총 문자열의 갯수 )

메모리의 사용량이 많이 든다. 적시적소에 잘 사용하여 단점을 줄이고 장점을 활용하자!!

구조

필자는 C++로 구현을 하였다.

struct node{

    bool finish ;

    node\* next[ALPHA];

}

각 노드는 26개의 노드 포인터와 finish 여부를 변수로 가진다.

함수 Insert

void insert(const char\* key){

        insert(key , root);

}

void insert(const char\* key, node\* next[]){

        int index = \*key -'A';

        if(!next[index]) next[index] = new node();

        if(\*(key + 1) == '\0'){

            next[index]->finish = true;

            return;

        }

        insert( key+1 , next[index]->next);

    }

텅 빈 Trie에 문자열들을 저장해주는 함수이다.

문자열 key를 매개 변수로 받아서 key의 문자를 따라 하나씩 이동 해가며 그에 해당하는 노드를 만들어 주는 함수이다. 이미 노드가 존재한다면 만들지 않는다.

이동할 때는 recursion을 이용한다.

문자의 끝에 도달했다면 finish를 true로 바꾸어 준다. ( Finish Defalut가 false 임 )

함수 find

문자열이 존재하는 지 찾아 주는 함수

bool find(const char\* key , node\* next[]){

        int index = \*key - 'A';

        if(!next[index]) return false;

        if(\*(key+1)=='\0'){

            if(next[index]->finish) return true;

            else return false;

        }

        return find(key+1 , next[index]->next);

    }

    bool find(const char\* key){

        return find(key, root);

    }

Insert와 코드가 비슷하다.

찾고자 하는 문자열 key를 하나씩 읽어가면서 노드를 이동해 간다.

이동하려는 노드가 없다면 false를 return

끝까지 도달했으나 finish가 false라면 마찬가지로 false를 return한다.

O(Log n) 의 시간 복잡도로 굉장히 빠르게 탐색 가능하다.

Trie를 응용하여 관련 검색 엔진을 만들어 보았다.

관련 검색 엔진은 네이버, 구글 등 대부분의 플랫폼에서 사용한다.

수많은 검색어들을 Trie를 이용하여 빠르게 탐색 한다면 관련 문자열도 빠르게 찾을 수 있다.

코드는 github에 올려두었다.

Github 주소 :

https://github.com/ohinhyuk/MCNL-Study/blob/main/%EA%B3%A0%EC%9C%A4%EB%AF%BC%EA%B5%90%EC%88%98%EB%8B%98%20%EC%8A%A4%ED%84%B0%EB%94%94/week%202/trie%EA%B8%B0%EB%B0%98\_%EA%B2%80%EC%83%89%EC%97%94%EC%A7%84%EA%B5%AC%ED%98%84/ver2/trie\_ver2.cpp

