트라이(trie)를 사용하여 문제 풀기

sianke1991

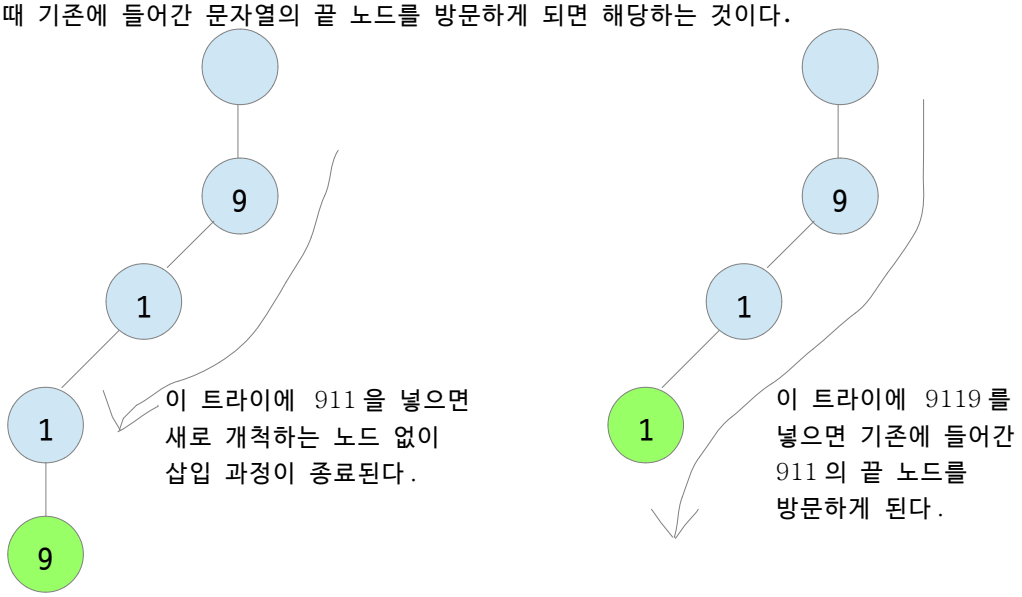
트라이는 이론상 시간복잡도는 훌륭하지만 문자열 집합에 삽입, 삭제, 조회하는데 시간이 오래 걸린다. (트리 혹은 해시보다 더 많은 시간을 소요한다.) 문자열 집합에 삽입, 삭제, 조회하는 문제를 푸는데 트라이를 사용하는 것은 좋은 전략이 아니다.
트라이는 검색엔진의 자동 완성 기능, 접두사/접미사와 관련된 문제를 푸는데 사용하는 것이 좋다.

#14426 접두사 찾기
문자열 집합 내에서 특정 문자열을 접두사로 갖는 문자열이 있는지 조회한다. 문자열 집합을 트라이에 넣고, 별도로 입력받은 접두사가 트라이에 있는지 여부를 조사하면 된다. 트라이에서 문자열을 조회하는 것과 달리 접두사를 조회할 때는 접두사의 끝이 반드시 특정 문자열의 끝과 일치할 필요가 없다는 것이다.

트라이에서 문자열 조회	트라이에서 접두사 조회

bool Find(string s) {
  int cur = ROOT;
  for (char ch:s) {
  if (nxt[cur][c2i(ch)]==-1) return false;
  cur = nxt[cur][c2i(ch)];
  }
  return chk[cur];
/*
마지막 문자까지 와서 해당 문자가 찾고자 하는 문자열의 끝 문자인지 확인해야 한다.
*/
}
	bool FindPrefix(string s) {
	  int cur = ROOT;
	  for (char ch:s) {
	  if (nxt[cur][c2i(ch)]==-1) return false;
	  cur = nxt[cur][c2i(ch)];
	  }
	  return true;
	/*
	마지막 문자까지 오면 그 자체로 입력받은 문자열을 접두사로 갖는 문자열이 트라이에 있는 것이다.
	*/
	}

#5052 전화번호 목록
문자열 집합에서 어떤 문자열이 다른 문자열의 접두사가 되는지 여부를 조사한다. 문자열 집합을 트라이에 넣을 때,
이번에 넣은 문자열이 트라이에 들어있는 어떤 문자열의 접두사가 되는지 혹은
이번에 넣은 문자열이 트라이에 들어있는 어떤 문자열을 접두사로 갖는지
여부를 조사한다. 전자는 트라이에 문자열을 넣을 때 새로 개척하는 노드가 없는 경우 해당하는 것이고, 후자의는 트라이에 문자열을 넣을 때 기존에 들어간 문자열의 끝 노드를 방문하게 되면 해당하는 것이다.



한 문자열이 다른 문자열의 접두사가 되는 두 경우
트라이에 문자열을 넣으면서 이 경우를 검출할 수 있다.

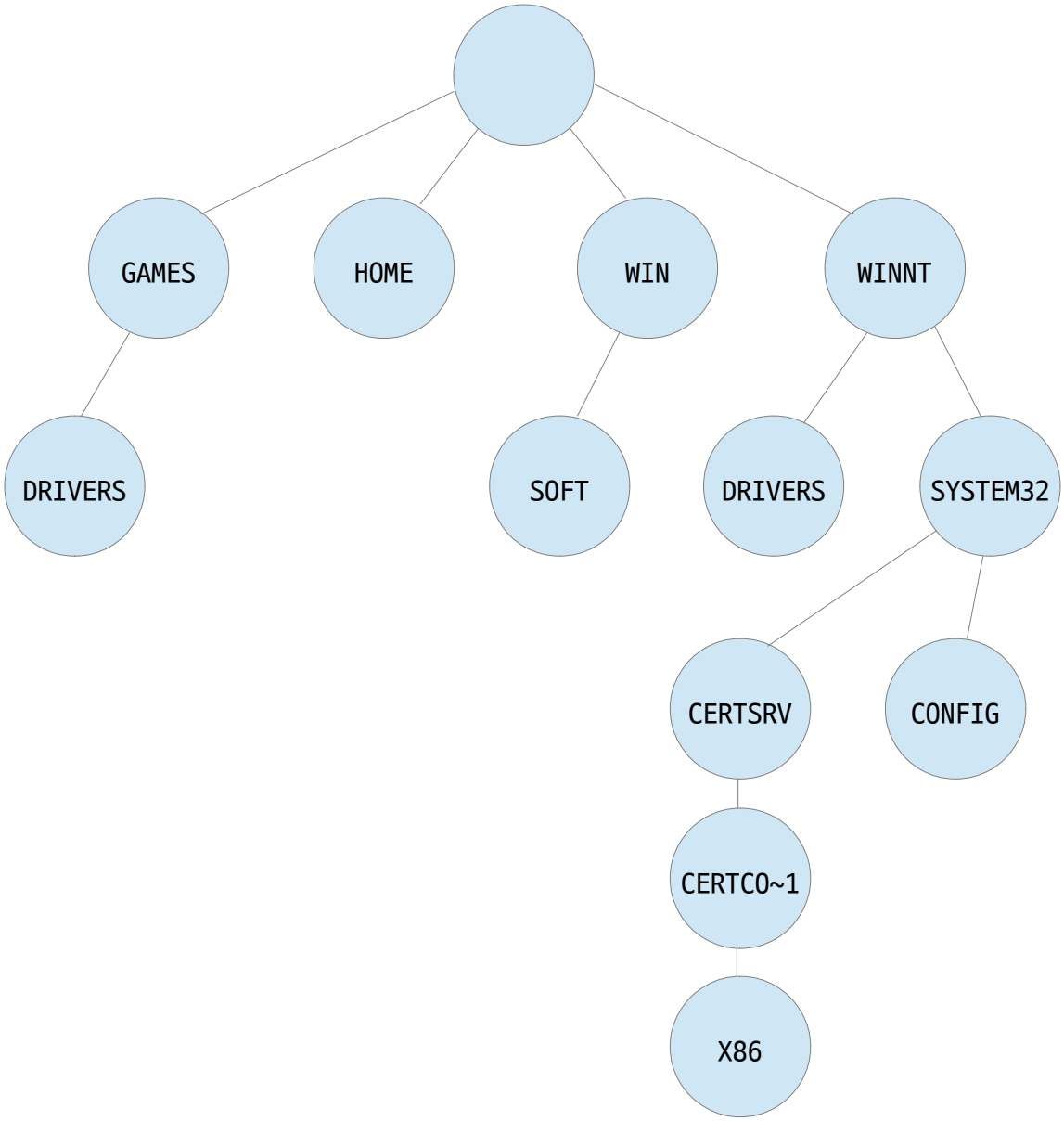
트라이에 문자열을 넣으면서 새로 넣을 문자열의 기존 문자열의 접두사가 되는가 혹은 기존 문자열을 접두사로 갖는가 여부를 판단하는 방법을 사용할 수 있다.

```
/**
 *  트라이에 문자열을 추가한다.<br/>
 *  일관성을 해치는 문자열이 들어오는 경우 false를 반환한다.
 */
/*
새로 입력된 전화번호가 일관성을 깨는 경우는 두 가지가 있다.
  1. 새로 입력된 전화번호가 기존에 있는 전화번호의 접두사가 되는 경우 (eg. 9119가 입력된 상태에서 911이 입력되는 경우)
  2. 새로 입력된 전화번호가 기존에 있는 전화번호 중 최소 하나를 접두사로 갖는 경우 (eg. 911이 입력된 상태에서 9119가 입력되는 경우)
  1번의 경우 새 전화번호를 트라이에 넣을 때 단 한번도 새로운 노드를 개척하지 않게 된다. 따라서 새 전화번호를 넣을 때 새 노드를 개척하는지 여부를 체크할 필요가 있다.
  2번의 경우 새 전화번호를 트라이에 넣을 때 기존에 입력된 단어의 끝을 방문하게 된다. 따라서 새 전화번호를 넣을 때 기존에 넣어진 단어의 끝을 방문하는지 여부를 체크할 필요가 있다.
*/
bool Insert(string s) {
    int cur = ROOT;
    bool result0 = false; //새로운 노드를 개척한 전적이 있는지 여부
    bool result1 = true; //기존에 입력된 단어의 끝을 만난 적이 없는지 여부
    for (char ch:s) {
        if (nxt[cur][c2i(ch)]==-1) { // 현재 보고 있는 노드에서 다음 문자에 해당하는 자식 노드가 없는 경우
            nxt[cur][c2i(ch)] = unused++; // 새로운 노드를 추가하여 현재 보고 있는 노드의 자식 노드로 삼는다.
            result0 = true; // 새로운 노드를 개척한 전적이 있는 경우 true로 설정한다. (이번에 입력된 단어는 기존에 입력된 단어들의 접두사가 아니다.)
        } else if (chk[nxt[cur][c2i(ch)]]) {
            result1 = false; //기존에 입력된 단어의 끝을 방문하게 되면 false로 반환한다. (이번에 입력된 단어는 기존에 입력된 단어 중 최소 하나를 접두사로 갖는다.)
        }
        cur = nxt[cur][c2i(ch)]; // 현재 보고 있는 노드를 자식 노드로 옮긴다.
    } //ch loop (트라이에 넣으려는 문자열의 각 문자에 대한 반복문)
    chk[cur] = true; //새로 추가하는 문자열의 마지막 문자에 해당하는 노드에 별도의 표시를 해 준다.
    return result0 && result1;
}
```

또는 다음 코드를 사용할 수 있다.

```
bool Insert2(string s) {
    int cur = ROOT;
    for (char ch:s) {
        if (nxt[cur][c2i(ch)]==-1) {
            nxt[cur][c2i(ch)] = unused++;
        }
        cur = nxt[cur][c2i(ch)];
        if (chk[cur]) return false; //기존에 삽입된 단어의 끝을 방문한 경우 바로 false를 반환한다.
    }
    if (cur!=unused-1) return false; //새로운 단어를 트라이에 삽입하면서 새 노드를 개척하지 않은 경우 false를 반환한다.
    chk[cur] = true;
    return true; //위 두 경우에 해당하지 않은 경우 트라이에 새로 추가된 문자열은 일관성을 깨지 않은 것이다.
}
//NOTE: unused++ 로 unused 값이 1 증가된 상태이므로 cur와 unused-1 값을 서로 비교한다.
```

#7432 디스크 트리
파일 경로들을 입력 받아서 디렉터리 구조를 보기 좋게 출력하는 프로그램을 만든다.



기존 문제들과 달리 이 트라이는 각 노드에 문자 하나가 아닌 문자열이 들어간다. (이렇게 구성해야 디렉터리 구조 출력이 편하다.) 또한 각 깊이에 맞게 앞에 공백 문자를 넣어서 트라이에 넣는 것이 편하다 (예를 들어 "SYSTEM32"가 아닌 " SYSTEM32"를 트라이에 넣는 식이다.)
이렇게 트라이를 구성했다면 디렉터리 구조 출력은 깊이 우선 탐색으로 쉽게 구현할 수 있다.

```
const int ROOT = 1;
int unused = ROOT;
const int MX = 500 * 41 + 5;
bool chk[MX];
string strArr[MX]; //해당 노드 번호에 들어간 문자열을 저장해준다. (트라이를 출력할 때 용이해짐)
map<string, int> nxt[MX];
```

입력받을 파일 경로의 디렉터리는 역슬래시로 구분한다. 이 파일 경로를 각 파일, 디렉터리로 구분하기 위해 하나의 문자열을 특정 문자로 나누는 함수를 구성하는 것이 편하다.

```
/**
 * 문자열을 입력받아 구분자를 기준으로 자른 결과를 반환한다.
 */
vector<string> split(string str, char delim) {
    vector<string> result;
    stringstream sstream(str);
    string word;
    while (getline(sstream, word, delim))
        result.push_back(word);
    return result;
}

/**
 * 입력받은 길이에 해당하는 공백 문자열을 반환한다.
 */
string whiteSpaces(int length) {
    string result = "";
    for (int i=0; i<length; i++)
        result += " ";
    return result;
}

void insert(string dir) {
    vector<string> words = split(dir, '\\');
    int cur = ROOT;
    int depth = 0; //해당 파일, 디렉터리의 깊이
    for (string word:words) {
        string newWord = whiteSpaces(depth)+word; //파일 또는 디렉터리의 이름 앞에 깊이에 해당하는 공백을 넣어서 문
자열 newWord를 구성한다.
        if (nxt[cur].find(newWord)==nxt[cur].end()) { //cur 노드가 newWord를 자식 노드로 가지고 있지 않으면-
            unused++;
            nxt[cur][newWord] = unused; //새로운 노드를 개척하고-
            strArr[unused] = newWord; //새로운 노드의 값을 별도로 보관해둔다.
        }
        cur = nxt[cur][newWord];
        depth++;
    } //word loop
    chk[cur] = true;
}

/**
 * 디렉터리 구조를 출력한다. (간단하게 재귀 호출로 구현할 수 있다.)
 */
void dfs(int cur) {
    cout << strArr[cur] << '\n';
    for (pair<string, int> entry:nxt[cur]) {
        dfs(entry.second);
    }
}
```

```
int main() {
    int N;
    cin >> N;
    string strInput;
    for (int i=0; i<N; i++) {
        cin >> strInput;
        insert(strInput);
    }
    /*
        단순히 dfs(ROOT) 호출 로도 디렉터리 구조는 출력이 되지만, ROOT 노드 출력 (빈 문자열)이 하나의 라인을
        차지하게 된다. 이 라인 출력을 막기 위해 dfs(ROOT)가 아닌 dfs(ROOT의 자식 노드들) 로 호출한다.
        (ROOT의 자식 노드들에 대해 dfs 함수를 호출하기 위해 for loop를 사용한다.)
    */
    for (pair<string, int> entry:nxt[ROOT]) {
        dfs(entry.second);
    }
    return 0;
}
```