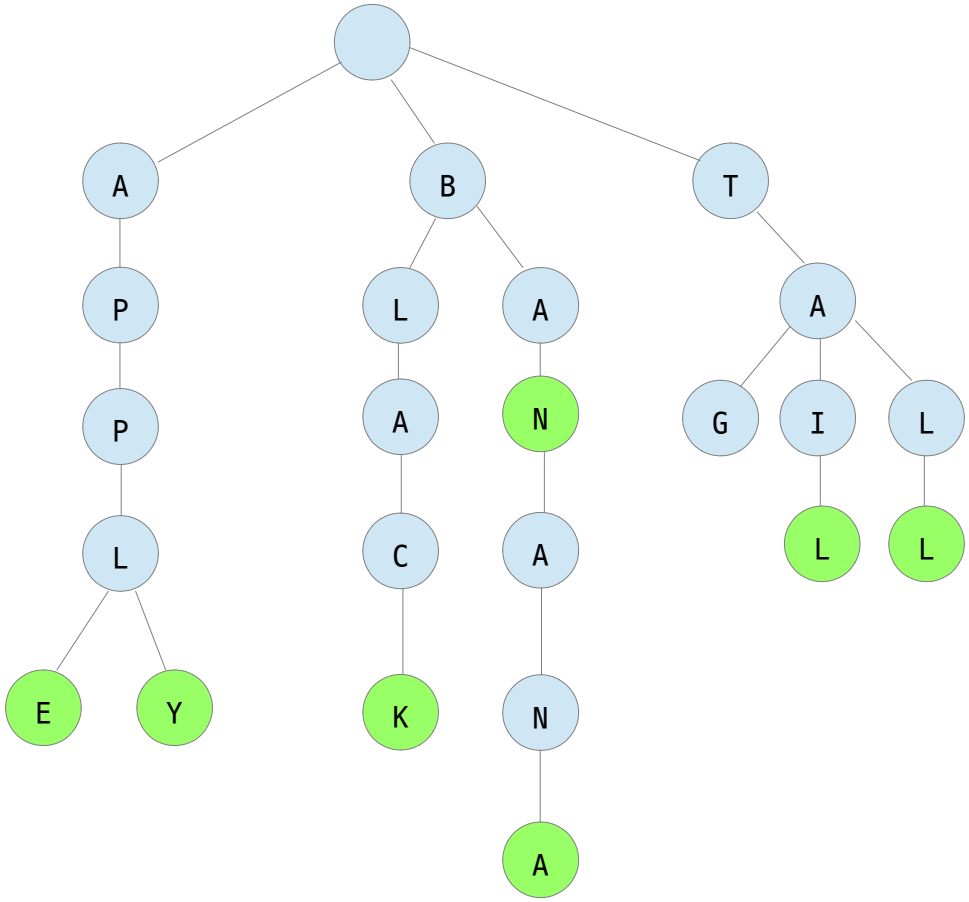
트라이(trie)

sianke1991

1. 트라이: 문자열을 효율적으로 처리하기 위한 자료 구조

시간복잡도 상 장점이 있고, 공간복잡도 상 단점이 있다.
장점: 단어 S를 삽입/ 탐색/ 삭제할 때 O(|s|)
단점: 문자열을 그냥 배열에 저장하는 것보다 최대 4*글자의 종류 배 만큼 더 사용 (여기서 4는 자료형 int의 크기)
(예를 들어 각 단어가 알파벳 대문자로만 구성되어 있을 경우 104배)

이론적인 시간복잡도와는 별개로 실제로는 트라이가 해시, 이진 검색 트리에 비해 훨씬 느림.
일반적인 상황에서는 해시나 이진 검색 트리를 사용하는 것이 좋으나,
트라이의 성질을 사용해야 하는 문제가 여럿 존재



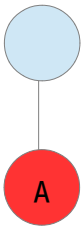
트라이의 예시. 단어 APPLE, APPLY, BLACK, BAN, BANANA, TAG, TAIL, TALL을 저장한 모습이다.
(루트 노드를 제외한) 각 노드는 문자 하나를 가지고 있고, 단어의 끝 문자인지 여부를 가지고 있다. (녹색으로 표시된 노드는 특정 단어의 끝 문자를 나타내는 노드이다.)
단어의 끝 문자가 반드시 리프 노드가 아님에 주의한다. 단어 BAN은 단어 BANANA의 부분문자열이므로, 단어 BAN은 리프 노드까지 뻗지 않고 중간에서 끝나버린다.
또한 이 트라이는 단어 APP을 저장하지 않음에 주의한다. APP의 두 번째 P 노드가 녹색이 아니므로 단어 APP은 저장하지 않은 것이다. (반대로 말하면 APP의 두 번째 P 노드를 녹색으로 변경하면 단어 APP을 저장하게 되는 것이고, BAN의 N 노드의 녹색을 제거하면 단어 BAN을 제거하는 것이다.)

2. 트라이에 단어 추가

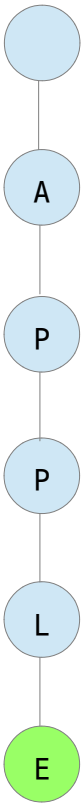
단어를 추가, 제거, 조회할 때는 루트 노드에서 시작하여 자식 노드로 차례대로 내려온다.

빈 트라이에 단어 APPLE 추가

루트에서 시작. (현재 보고 있는 노드는 붉은 색으로 표기한다.)

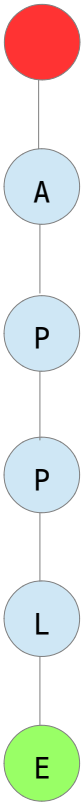


현재 보고 있는 노드에 문자 A를 갖는 자식 노드가 없으므로 자식 노드를 추가하고,
현재 보고 있는 노드를 한 단계 내린다.

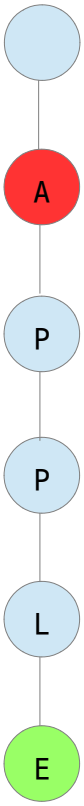


이 과정을 반복하여 마지막 문자 E 까지 추가하면 과정을 종료한다.
(마지막 문자 E는 별도의 표시 – 이 그림에서는 녹색으로 표시 – 를 하고 추가한다.)

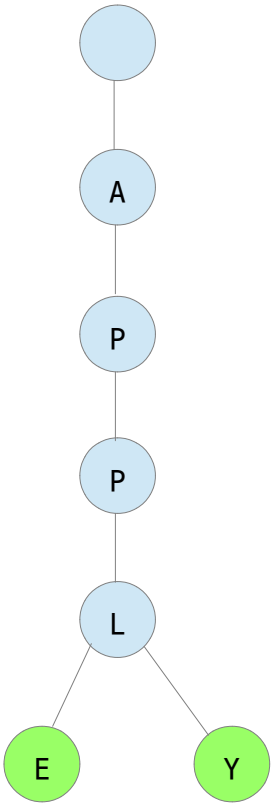
단어 APPLE이 있는 트라이에 단어 APPLY 추가



루트에서 시작

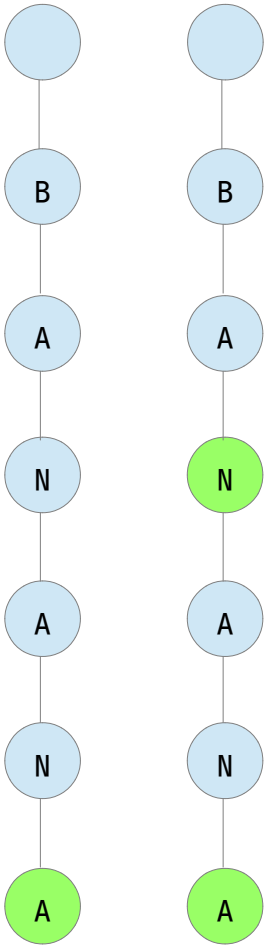


현재 보고 있는 노드에 문자 A를 갖는 자식 노드가 있으므로 노드 추가 없이 현재 보고 있는 노드를 한 단계 내린다.



L 노드까지 추가 없이 내려온 뒤,
(L 노드에 Y 자식 노드가 없었기 때문에) Y 노드를 추가하면 과정을 종료한다.
(마찬가지로 마지막 문자 Y는 별도의 표시를 하고 추가한다.)

단어 BANANA가 있는 트라이에 단어 BAN 추가



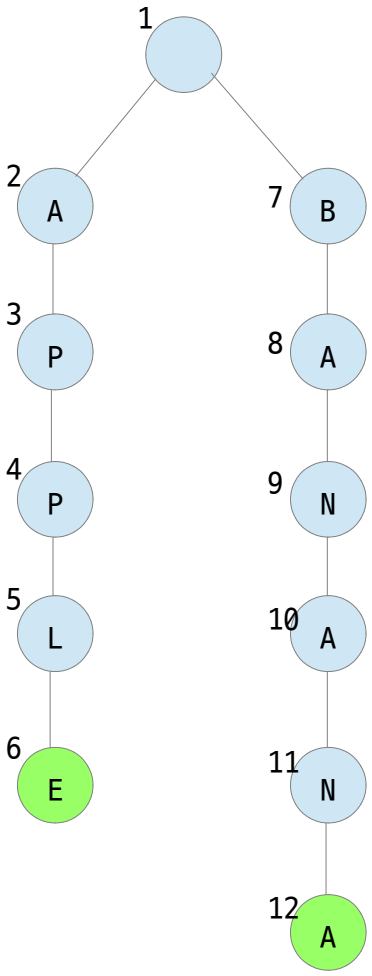
단어 BANANA가 있는 트라이에 단어 BAN을 추가할 때-
별도로 추가되는 노드는 없다. 이 경우 BAN의 N 노드에 별도의 표시를 추가하고
과정이 종료된다.

코드로 살펴보기

```
const int ROOT = 1; //루트 노드의 번호 (이후에 노드가 추가되면 추가된 노드는 각각 2, 3, 4 등의 번호를 갖는다.)
int unused = 2; //새로 추가될 노드의 번호 (2번 노드를 생성한 경우 unused++;를 수행하여 unused를 3으로 변경시켜둔다.)
const int MX = 10000 * 500 + 5; // 최대 등장 가능한 글자수
bool chk[MX]; //해당 노드가 문자열의 끝을 나타내는지 여부
int nxt[MX][26]; //해당 노드의 자식 노드
                //한 노드 당 자식 노드를 최대 26개 둘 수 있으므로 각 노드 당 길이 26짜리 배열을 두어 자식 노드를 관리한다.
                //이 방식은 자식 노드 탐색을 빠르게 해 주나 메모리 낭비가 심하다.

/**트라이 내 모든 노드에 자식 노드가 없는 상태로 만든다.*/
void init() {
    for (int i=0; i<MX; i++) {
        for (int j=0; j<26; j++) {
            nxt[i][j] = -1;
        }
    }
}
```

```
/**로마자 대문자 A-Z를 정수 0-25에 대응시켜 인덱스 값으로 활용할 수 있도록 만든다.*/
int c2i(char ch) {
    return ch-'A';
}
```



이 예시에서

```
unused = 13 //새로 추가될 노드가 있다면 그 노드는 13번이 된다.
nxt[1][c2i('A')] = 2 //1번 노드의 자식 노드 중 'A'를 갖는 노드는 2번 노드이다.
nxt[1][c2i('B')] = 7 //1번 노드의 자식 노드 중 'B'를 갖는 노드는 7번 노드이다.
nxt[1][c2i('C')] = -1 //1번 노드의 자식 노드 중 'C'를 갖는 노드는 없다.
chk[6] = true, chk[12] = true, chk[5] = false //6번, 12번 노드는 단어의 끝 문자를 저장하는 노드이지만, 5번 노드는 단어의 끝 문자를 저장하는 노드가 아니다.
```

```
/**트라이에 문자열을 추가한다.*/
void Insert(string s) {
    int cur = ROOT; //현재 보고 있는 노드
    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.
        }
        cur = nxt[cur][c2i(ch)]; //현재 보고 있는 노드를 한 단계 내린다.
    } //ch loop (트라이에 넣으려는 문자열의 각 문자에 대한 반복문)
    chk[cur] = true; //새로 추가하는 문자열의 마지막 문자에 해당하는 노드에 별도의 표시를 해 준다.
}
```

3. 트라이에 단어 검색

트라이에 특정 단어가 있는지 여부를 검색한다.

```
bool Find(string s) {
    int cur = ROOT;
    for (char ch:s) {
        if (nxt[cur][c2i(ch)]==-1) return false; //찾고자 하는 단어의 마지막 문자까지 내려가지 않았는데 노드가 없는 경우 트라이에는 해당 문자열이 없다고 결론 내린다.
        cur = nxt[cur][c2i(ch)]; //찾고자 하는 문자가 있는 경우 현재 보고 있는 노드를 한 단계 내려서 검색을 계속 한다.
    } //ch loop (트라이에서 검색하려는 문자열의 각 문자에 대한 반복문)
    return chk[cur]; //찾고자 하는 단어의 마지막 문자에 해당하는 노드에 별도의 표시가 되어 있어야 트라이에 해당 단어가 있는 것이다.
}
```

4. 트라이에서 단어 제거

트라이에 특정 단어를 제거하려면 단어의 마지막 문자 노드에 해 놓는 표시를 지우는 방식을 사용한다. (함부로 노드를 제거하면 다른 단어에 영향을 미칠 수 있다. 단어 BAN을 제거하기 위해 N 노드를 제거한 경우 단어 BANANA 까지 추가로 제거될 수 있다.)

```
void Remove(string s) {
    int cur = ROOT;
    for (char ch:s) {
        if (nxt[cur][c2i(ch)]==-1) return; //제거하고자 하는 단어가 트라이에 없음이 확실한 경우
                                          //이 경우 과정을 바로 중단할 수 있다.
                                          //만약 제거하고자 하는 단어가 트라이에 있는 것이 확실한 경우
                                          //이 if 문을 생략할 수 있다.
        cur = nxt[cur][c2i(ch)];
    }
    chk[cur] = false; //단어의 마지막 문자 노드에 있는 표시를 지운다.
}
```

5. 메모리 절약 방법: 자식 노드를 저장하기 위해 int nxt[MX][26]을 사용하면 자식 노드 탐색은 빠르게 수행하지만 메모리 낭비가 크다. 자식 노드 탐색의 속도를 낮추더라도 메모리 낭비를 줄이기 위해서는 다른 방법으로 자식 노드를 저장하면 된다. 다음과 같은 방식으로 자식 노드를 저장할 수 있다.

BOJ14425 문제 제한 조건을 바탕으로 계산

자식 노드 저장 방법	코드	시간복잡도	공간복잡도	시간	메모리
고정된 크기의 배열	nxt[MX][26]	O(¦S¦)	O(26·MX)	472 ms	514 MB
동적 배열	vector<pair<char, int>> nxt[MX]	O(26·¦S¦)	O(MX)	2018 ms	280 MB
연결 리스트	list<pair<char, int>> nxt[MX]	O(26·¦S¦)	O(MX)	3732 ms	280 MB
이진 검색 트리	map<char, int> nxt[MX]	O(log(26)·¦S¦)	O(MX)	688 ms	474 MB
해시	unordered_map<char, int> nxt[MX]	O(¦S¦)	O(MX)	1584 ms	979 MB

배열 대신 이진 검색 트리를 사용하는 경우 추가, 검색, 제거 코드는 다음과 같이 변경된다.

```
const int ROOT = 1;
int unused = 2;
const int MX = 10000 * 500 + 5;
bool chk[MX];
map<char, int> nxt[MX];

void Insert(string s) {
    int cur = ROOT;
    for (char ch:s) {
        if (nxt[cur].find(ch)==nxt[cur].end()) { //cur 노드에 ch를 갖는 자식 노드가 없는 경우
            nxt[cur][ch] = unused++;
        }
        cur = nxt[cur][ch];
    }
    chk[cur] = true;
}

bool Find(string s) {
    int cur = ROOT;
    for (char ch:s) {
        if (nxt[cur].find(ch)==nxt[cur].end()) return false;
        cur = nxt[cur][ch];
    }
    return chk[cur];
}
```

```
void Remove(string s) {
    int cur = ROOT;
    for (char ch:s) {
        if (nxt[cur].find(ch)==nxt[cur].end()) return;
        cur = nxt[cur][ch];
    }
    chk[cur] = false;
}
```