Sogang ICPC Team

2021-1 중급 스터디 5회차

임지환
raararaara@gmail.com
2021.5.9

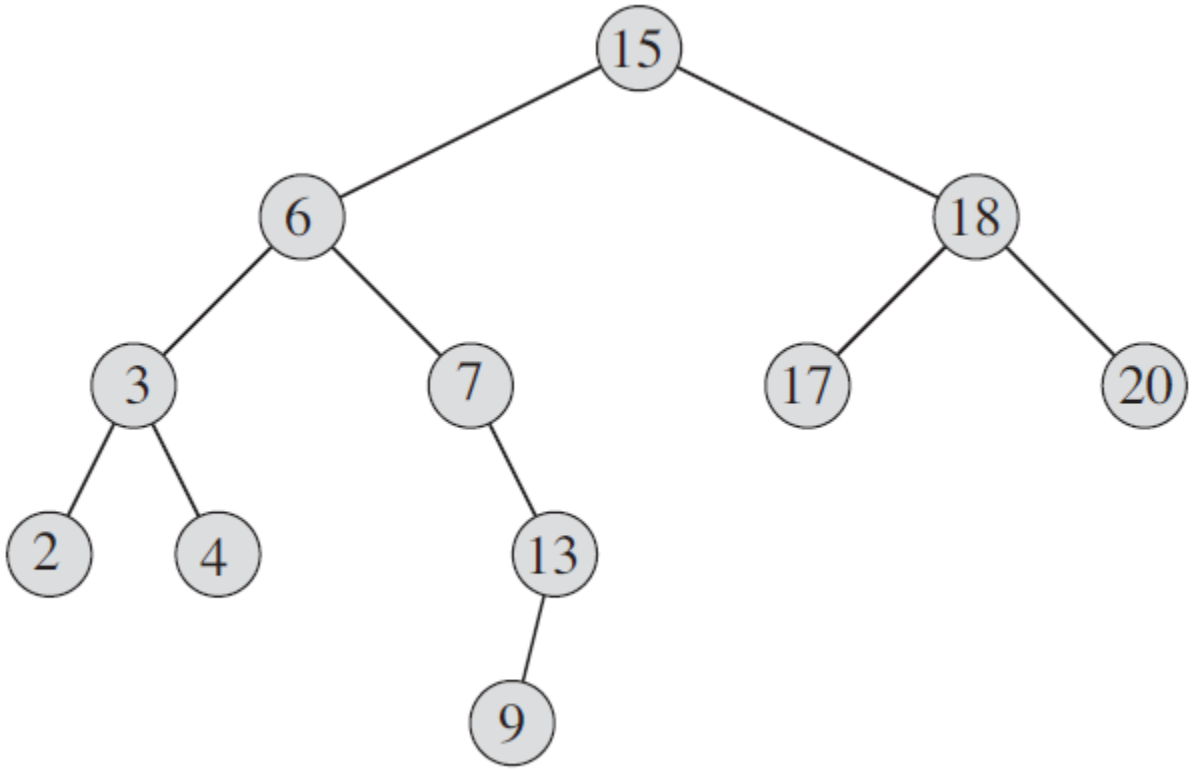
목차

- STL containers based on BST
  - Binary Search Tree
  - std::set & std::map
- Hashing
  - Polynomial rolling hash function
  - Handling Collision
  - Rabin-Karp pattern matching algorithm
- Trie
  - Basic Concept
  - Binary Trie

STL containers based on BST

- Binary Search Tree
- std::set & std::map

Binary Search Tree



- N개의 노드로 구성된 Balanced BST의 경우:

✓트리의 깊이가 $\log N$에 근사
✓탐색 및 삽입, 삭제에 $O(\log N)$

- N개의 노드로 구성된 Balanced BST의 경우:

✓트리의 깊이가 $\log N$에 근사

✓탐색 및 삽입, 삭제에 $O(\log N)$

정말 그럴까?

- N개의 노드에 string을 담고 있다면?

- N개의 노드에 string을 담고 있다면?

✓BST를 따라 탐색을 하는 과정에서 만나는 노드 개수:

- N개의 노드에 string을 담고 있다면?

✓BST를 따라 탐색을 하는 과정에서 만나는 노드 개수: $\log N$

- N개의 노드에 string을 담고 있다면?

✓BST를 따라 탐색을 하는 과정에서 만나는 노드 개수: $\log N$

✓탐색을 할 때 left child, 혹은 right child와의 비교 횟수:

Binary Search Tree

- N개의 노드에 string을 담고 있다면?

✓BST를 따라 탐색을 하는 과정에서 만나는 노드 개수: $\log N$

✓탐색을 할 때 left child, 혹은 right child와의 비교 횟수: $\min(|P|, |C|)$

$$where\ P: string\ of\ parent,$$

$$C: string\ of\ child$$

✓길이를 M이라 하면, 대략 $O(M \log N)$

- std::set

✓Balanced BST 중 하나인 red-black tree 기반으로 구현된 탐색 자료구조
✓중복(duplicate)을 허용하지 않는, 문자 그대로 집합(set)
✓삽입, 삭제, 검색 모두 $O(\log N)$
✓중복 허용이 되지 않으니 존재 유무 정도로 사용 가능

- std::map

✓Balanced BST 중 하나인 red-black tree 기반으로 구현된 탐색 자료구조

✓Key-Value 쌍으로 저장

✓각 Key를 unique하게 관리, 즉 탐색을 Key기준으로.

✓삽입, 삭제, 검색 모두 $O(\log N)$

✓set과 달리, value값 조정을 통해

1) 원소의 등장 횟수
2) Key값에 대한 labeling이 가능

- std::map

✓Balanced BST 중 하나인 red-black tree 기반으로 구현된 탐색 자료구조

✓Key-Value 쌍으로 저장

✓각 Key를 unique하게 관리, 즉 탐색을 Key기준으로.

✓삽입, 삭제, 검색 모두 $O(\log N)$

✓set과 달리, value값 조정을 통해

1) 원소의 등장 횟수 ⇒ `std::multiset`으로 원소 개수 관리 가능
2) Key값에 대한 labeling이 가능

#1764 듣보잡

- 듣도 못한 사람의 수 N, 보도 못한 사람의 수 M $(1 \leq N, M \leq 5 \times 10^5)$
- 이름의 길이가 20이하

- 듣보잡 == 듣도 보도 못한 사람

#1764 듣보잡

- 듣도 못한 사람 중 보도 못한 사람의 목록에도 껴 있는지 확인을 하자.

```cpp
int N, M;
map<string, int> d;
int main() {
    cin >> N >> M;
    for (int i = 0; i < N; i++) {
        string s;
        cin >> s;
        if(d.find(s) == d.end())
            d.insert({ s, 1 });
    }
    int ans = 0;
    set<string> st;
    for (int i = 0; i < M; i++) {
        string s;
        cin >> s;
        if (d.find(s) != d.end()) ans++, st.insert(s);
    }
    cout << ans << '\n';
    for (string it : st) cout << it << '\n';
    return 0;
}
```

Hashing

- Polynomial rolling hash function
- Handling Collision
- Rabin-Karp pattern matching

Polynomial rolling hash function

- string으로 비교 탐색 시 $O(M \log N)$

✓string → integer로 변환할 수 있다면?⇒ $O(\log N)$

Polynomial rolling hash function

- 소문자 알파벳으로만 구성된 단어가 주어진다면:

a=1 b=2 c=3 ..... x=24 y=25 z=26

apple $\Rightarrow 1 \times 27^4 + 16 \times 27^3 + 16 \times 27^2 + 12 \times 27^1 + 5 \times 27^0 = 858362$

car $\Rightarrow 3 \times 27^2 + 1 \times 27^1 + 18 \times 27^0 = 2232$

6글자까지는 `int`범위 내에, 13글자까지는 `long long` 범위 내에 표현 가능

#2866 문자열 잘라내기

- R개의 행과 C개의 열로 이루어진 테이블 $(2 \leq R, C \leq 1{,}000)$
- 테이블의 각 열을 위에서 아래로 읽어서 열 간 문자열이 중복되지 않는다면 가장 위의 행을 삭제
- 최초 테이블은 동일한 문자열이 존재하지 않음(count변화없이 지운다고 가정)
- 몇 행까지 삭제할 수 있을까?

#2866 문자열 잘라내기

1. 크기가 최대 1000×1000
2. 각 행으로 시작하는 접미사(suffix)문자열의 hash값을 배열에 저장하자.

S	O	G
A	N	G
B	A	B
A	B	A

$27^3S + 27^2A + 27B + A$	$27^3O + 27^2N + 27A + B$	$27^3G + 27^2G + 27B + A$
$27^2A + 27B + A$	$27^2N + 27A + B$	$27^2G + 27B + A$
$27B + A$	$27A + B$	$27B + A$
A	B	A

#2866 문자열 잘라내기

1. 크기가 최대 1000×1000
2. 각 행으로 시작하는 접미사(suffix)문자열의 hash값을 배열에 저장하자.

S	O	G
A	N	G
B	A	B
A	B	A

$27^3S + 27^2A + 27B + A$	$27^3O + 27^2N + 27A + B$	$27^3G + 27^2G + 27B + A$
$27^2A + 27B + A$	$27^2N + 27A + B$	$27^2G + 27B + A$
$27B + A$	$27A + B$	$27B + A$
A	B	A

#2866 문자열 잘라내기

```
 8    int N, M;
 9    string s[1001];
10    lint h[1001][1001];
11
12    lint p = 27;
13
14    int main() {
15        cin >> N >> M;
16        for (int i = 0; i < N; i++)
17            cin >> s[i];
18        lint pp = 1;
19        for (int r = N - 1; r >= 0; r--, pp = pp * p)
20            for (int c = 0; c < M; c++)
21                h[r][c] = h[r + 1][c] + (s[r][c] - 'a' + 1) * pp;
22        for (int i = 1; i < N; i++) {
23            sort(h[i], h[i] + M);
24            for (int j = 0; j < M - 1; j++) if (h[i][j] == h[i][j + 1])
25                return cout << i - 1, 0;
26        }
27        cout << N - 1;
28        return 0;
29    }
```

#9 s[r] : r번째 행의 정보

#10 h[r][c] : s[r][c]로 끝나는 접미사의 hash value

#11 p : base

#2866 문자열 잘라내기

```
 8   int N, M;
 9   string s[1001];
10   lint h[1001][1001];
11
12   lint p = 27;
13
14   int main() {
15       cin >> N >> M;
16       for (int i = 0; i < N; i++)
17           cin >> s[i];
18       lint pp = 1;
19       for (int r = N - 1; r >= 0; r--, pp = pp * p)
20           for (int c = 0; c < M; c++)
21               h[r][c] = h[r + 1][c] + (s[r][c] - 'a' + 1) * pp;
22       for (int i = 1; i < N; i++) {
23           sort(h[i], h[i] + M);
24           for (int j = 0; j < M - 1; j++) if (h[i][j] == h[i][j + 1])
25               return cout << i - 1, 0;
26       }
27       cout << N - 1;
28       return 0;
29   }
```

#19 이전 suffix값 으로부터 누적을 위해 역순으로 진행

#21 누적 값 + $27^k \times c$

#2866 문자열 잘라내기

```
 8   int N, M;
 9   string s[1001];
10   lint h[1001][1001];
11
12   lint p = 27;
13
14   int main() {
15       cin >> N >> M;
16       for (int i = 0; i < N; i++)
17           cin >> s[i];
18       lint pp = 1;
19       for (int r = N - 1; r >= 0; r--, pp = pp * p)
20           for (int c = 0; c < M; c++)
21               h[r][c] = h[r + 1][c] + (s[r][c] - 'a' + 1) * pp;
22       for (int i = 1; i < N; i++) {
23           sort(h[i], h[i] + M);
24           for (int j = 0; j < M - 1; j++) if (h[i][j] == h[i][j + 1])
25               return cout << i - 1, 0;
26       }
27       cout << N - 1;
28       return 0;
29   }
```

#22~26 중복이 존재하면 종료

#27 중복이 나오지 않는 경우

Polynomial rolling hash function

- `map || set` 써도 되겠는데요?

✓적절한 소수 (ex: $p = 10^9 + 7$)로 나눈 나머지만을 저장하면 겹칠 확률이 $\frac{1}{p}$이므로 안전

Polynomial rolling hash function

- `map || set` 써도 되겠는데요?

✓적절한 소수 (ex: $p = 10^9 + 7$)로 나눈 나머지만을 저장하면 겹칠 확률이 $\frac{1}{p}$이므로 안전

정말 그럴까?

Collision

- Birthday Conjecture: 1년 365일에서, 임의로 23명을 고른 경우 그 중에 **생일이 겹치는 경우가 있을 확률이 50%**

✓겹칠 확률 $p(n) \approx \frac{n^2}{2m}$: m은 1년의 날 수, n은 사람 수

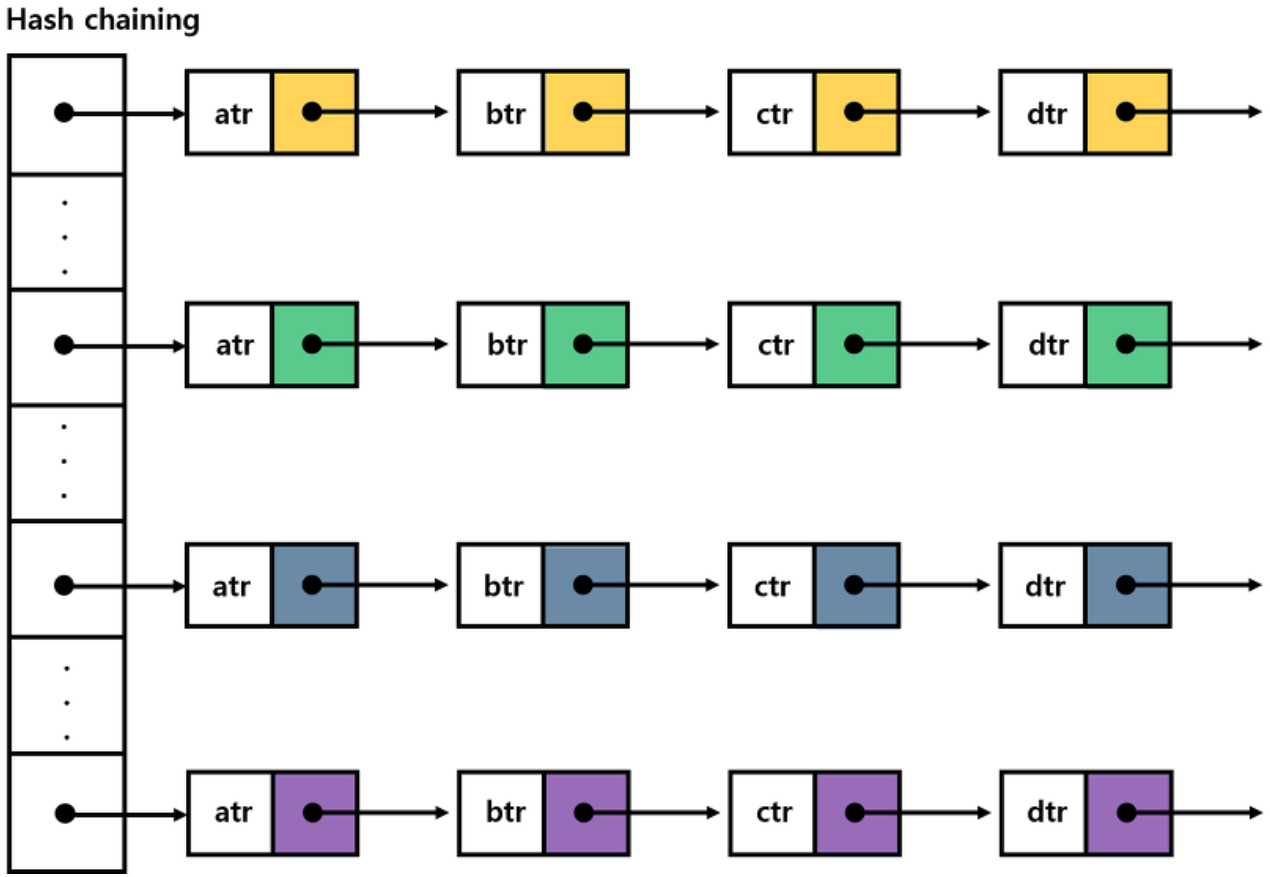
✓같은 논리로, 해싱 처리된 문자열의 개수가 $\sqrt{10^9} \approx 3 \times 10^4$개만 되어도 50%확률로 겹침ㅠ

Hash Collision

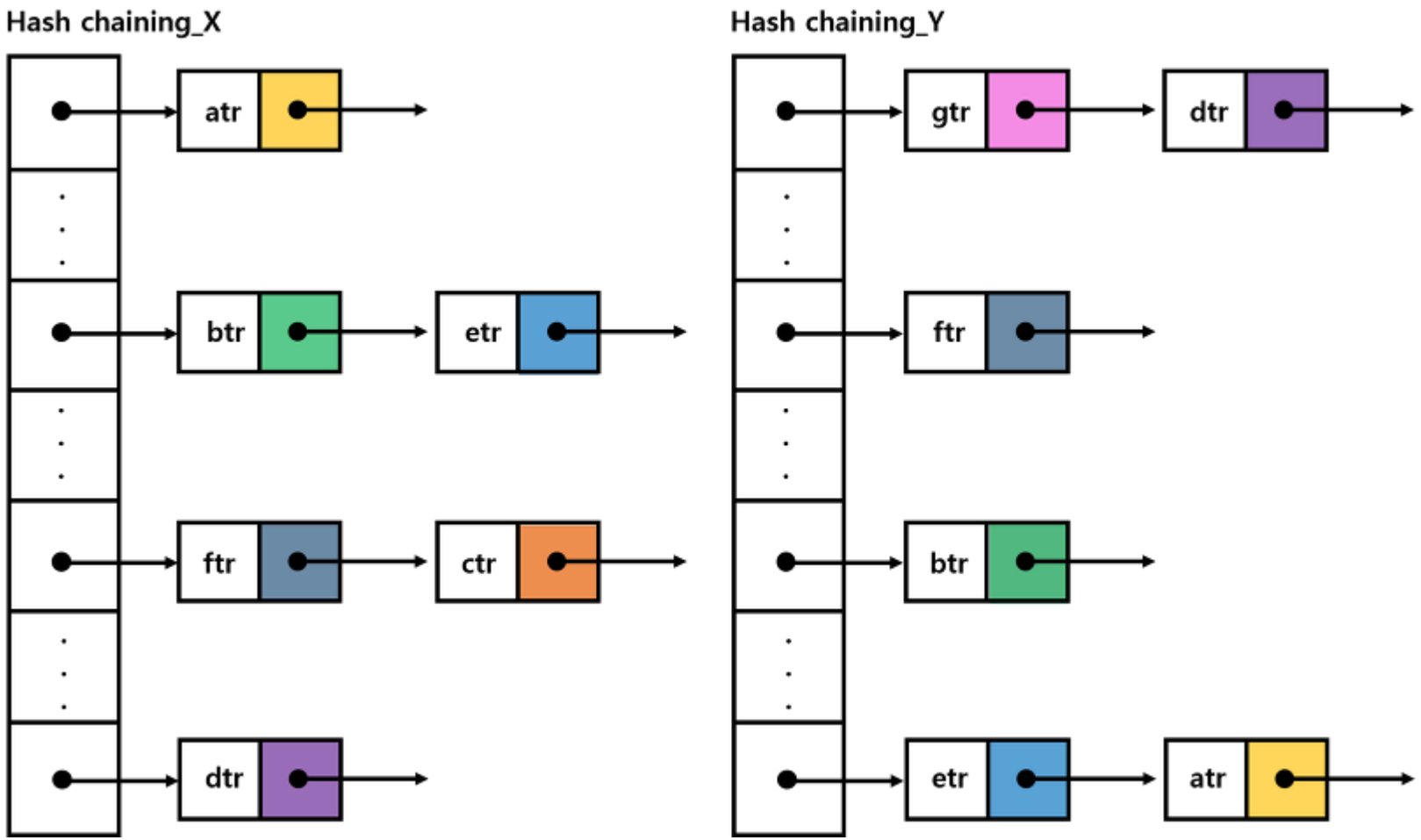
- Handling Collision

1) Chaining
2) 원본 문자열을 같이 저장
3) 2개의 hash 이용

- Chaining


Hash chaining
atr
btr
ctr
dtr
atr
btr
ctr
dtr
atr
btr
ctr
dtr
atr
btr
ctr
dtr

- 2개의 hash 이용


Hash chaining_X
atr
btr
etr
ftr
ctr
dtr
Hash chaining_Y
gtr
dtr
ftr
btr
etr
atr

Collision

- 2개의 hash 이용

✓얼마나 작아질까?

- $10^9 + 7$과 $10^9 + 9$로 해싱한 값을 둘 다 들고 있으면 문자열 개수 3×10^4에서 collision이 일어날 확률은 약 0.000000045%
- 문자열 개수 10^6에서도 확률은 약 0.005%

✓0%는 아니다ㅠ

#10840 구간 성분

- 길이가 N, M인 두 문자열 $(1 \leq N, M \leq 1{,}500)$
- 어떤 구간에 포함된 문자의 종류와 개수가 순서에 상관없이 동일한 경우 '구간의 성분이 같다'고 표현

- 두 문자열에서 같은 성분을 가진 구간 중 가장 긴 구간의 길이

a f c d r d e s d e f w s z r

g e d s r d d e m z r

#10840 구간 성분

- 같은 구간 성분의 조건
  1) 구간 길이가 같아야 함
  2) 구성(알파벳 당 등장 횟수)이 같아야 함

- 최적 해(=M)보다 작은 L에 대하여, 길이가 L인 동일 구간 성분이 존재한다고 할 때 L+1도 가능한가? (=단조성이 성립하는가?)

#10840 구간 성분

- 같은 구간 성분의 조건
  1) 구간 길이가 같아야 함
  2) 구성(알파벳 당 등장 횟수)이 같아야 함

- 최적 해(=M)보다 작은 L에 대하여, 길이가 L인 동일 구간 성분이 존재한다고 할 때 L+1도 가능한가? (=단조성이 성립하는가?)

NO

#10840 구간 성문

- 문제 해결 과정

1. 성분과 구성은 독립이므로 문자의 index를 기준으로 rolling hash를 구성하지 않고 알파벳 별로 구성

   $[l,r]$구간에서 나오는 알파벳 개수 각각을 $cnt[i]$라 할 때

$$hash(l,r) = \sum_{i=0}^{25} \left(cnt[i] \times p^i \right)$$

1. 각 길이에 대하여 같은 구성 성분인 pattern이 등장하는지 확인

#10840 구간 성분

- Time Complexity Analysis

✓각 길이에 대하여 ⇒ $O(N)$

✓두 서열 상에서 매칭되는 구간이 있는지 확인 ⇒ $O(MN)$

$$O(N^3) \Rightarrow TLE$$

#10840 구간 성분

- Time Complexity Analysis

✓각 길이에 대하여 ⇒ $O(N)$

✓두 서열 상에서 매칭되는 구간이 있는지 확인 ⇒ ~~$O(MN)$~~ $O(M \log N)$

$$O(N^3) \Rightarrow O(N^2 \log N)$$

#10840 구간 성분

```
14  int f(int len) {
15      memset(cnt, 0, sizeof cnt);
16      map<lint, int> mp;
17      lint v = 0;
18      for (int i = 0; i < len; i++)
19          v += pw[s[0][i] - 'a'];
20      mp[v]++;
21      for (int i = len; i < N; i++) {
22          v -= pw[s[0][i - len] - 'a'];
23          v += pw[s[0][i] - 'a'];
24          mp[v]++;
25      }
26      v = 0;
27      memset(cnt, 0, sizeof cnt);
28      for (int i = 0; i < len; i++)
29          v += pw[s[1][i] - 'a'];
30      if (mp.find(v) != mp.end()) return 1;
31      for (int i = len; i < M; i++) {
32          v -= pw[s[1][i - len] - 'a'];
33          v += pw[s[1][i] - 'a'];
34          if (mp.find(v) != mp.end()) return 1;
35      }
36
37      return 0;
38  }
```

#18~25 앞의 문자열에서 길이가 len인 부분문자열의 hash value 저장

#18~20 최초 접두사의 hash

#22~23 왼쪽 부분의 out of range index는 빼고, 오른쪽 부분의 추가되는 index는 더하기

#10840 구간 성분

```
14  int f(int len) {
15      memset(cnt, 0, sizeof cnt);
16      map<lint, int> mp;
17      lint v = 0;
18      for (int i = 0; i < len; i++)
19          v += pw[s[0][i] - 'a'];
20      mp[v]++;
21      for (int i = len; i < N; i++) {
22          v -= pw[s[0][i - len] - 'a'];
23          v += pw[s[0][i] - 'a'];
24          mp[v]++;
25      }
26      v = 0;
27      memset(cnt, 0, sizeof cnt);
28      for (int i = 0; i < len; i++)
29          v += pw[s[1][i] - 'a'];
30      if (mp.find(v) != mp.end()) return 1;
31      for (int i = len; i < M; i++) {
32          v -= pw[s[1][i - len] - 'a'];
33          v += pw[s[1][i] - 'a'];
34          if (mp.find(v) != mp.end()) return 1;
35      }
36
37      return 0;
38  }
```

#28~35 아래 문자열에서 길이가 len인 부분문자열 중 같은 구성의 부분문자열 찾기

Rabin-Karp Algorithm

- Handling Collision

1) Chaining
2) **원본 문자열을 같이 저장**
3) 2개의 hash 이용

- Handling Collision

1) Chaining
2) **원본 문자열을 같이 저장 ⇒ 첫 index만 저장, hash값이 같을 경우 단순 비교**
3) 2개의 hash 이용

- Handling Collision

1) Chaining
2) 원본 문자열을 같이 저장 ⇒ 첫 index만 저장, hash값이 같을 경우 단순 비교
3) 2개의 hash 이용 단순 비교보다는 double hash가 빠름ㅠ

#1786 찾기

- 문장 하나, 패턴 문장 하나($1 \leq |S|, |M| \leq 10^6$)
- 패턴이 문장 안에 있는지, 있으면 그 개수와 위치를 구하여라.

- 문제 해결 과정

✓Hash Collision이 일어날 확률이 적긴 하지만, 100만 시간의 단순비교는 느리다.
✓Double hash를 활용해보자.

#1786 찾기

```
16      vector<int> ans;
17      const lint p = 31, MOD[2] = { (lint)1e9 + 7, (lint)1e9 + 9 };
18      lint h[2] = { 0,0 }, hh[2] = { 0,0 }, pw[2] = { 1, 1 };
19
20      for (int i = 0; i < M; i++) {
21          for (int j = 0; j < 2; j++) {
22              h[j] = (h[j] * p + pat[i]) % MOD[j];
23              hh[j] = (hh[j] * p + s[i]) % MOD[j];
24              if (i) pw[j] = (pw[j] * p) % MOD[j];
25          }
26      }
27
28      if (h[0] == hh[0] && h[1] == hh[1]) ans.push_back(1);
29      for (int i = 1; i + M - 1 < N; i++) {
30          for (int j = 0; j < 2; j++) {
31              hh[j] = (p * (hh[j] - pw[j] * s[i - 1]) + s[i + M - 1]) % MOD[j];
32              if (hh[j] < 0) hh[j] += MOD[j];
33          }
34          if (h[0] == hh[0] && h[1] == hh[1]) ans.push_back(i + 1);
35      }
36
37      printf("%d\n", (int)ans.size());
38      for (int&it : ans) printf("%d ", it);
39
40      return 0;
41  }
```

s[]: 문장, pat[]: 찾고자 하는 패턴

#18

h[]: pat[]에서 등장하는 hash value

hh[]: s[]에서 등장하는 hash value

pw[]: $p^{m-1} \ mod \ MOD_i$

#1786 찾기

```
16      vector<int> ans;
17      const lint p = 31, MOD[2] = { (lint)1e9 + 7, (lint)1e9 + 9 };
18      lint h[2] = { 0,0 }, hh[2] = { 0,0 }, pw[2] = { 1, 1 };
19
20      for (int i = 0; i < M; i++) {
21          for (int j = 0; j < 2; j++) {
22              h[j] = (h[j] * p + pat[i]) % MOD[j];
23              hh[j] = (hh[j] * p + s[i]) % MOD[j];
24              if (i) pw[j] = (pw[j] * p) % MOD[j];
25          }
26      }
27
28      if (h[0] == hh[0] && h[1] == hh[1]) ans.push_back(1);
29      for (int i = 1; i + M - 1 < N; i++) {
30          for (int j = 0; j < 2; j++) {
31              hh[j] = (p * (hh[j] - pw[j] * s[i - 1]) + s[i + M - 1]) % MOD[j];
32              if (hh[j] < 0) hh[j] += MOD[j];
33          }
34          if (h[0] == hh[0] && h[1] == hh[1]) ans.push_back(i + 1);
35      }
36
37      printf("%d\n", (int)ans.size());
38      for (int&it : ans) printf("%d ", it);
39
40      return 0;
41  }
```

#20~26 s와 pat의 최초 접두사의 hash value

#1786 찾기

```
16      vector<int> ans;
17      const lint p = 31, MOD[2] = { (lint)1e9 + 7, (lint)1e9 + 9 };
18      lint h[2] = { 0,0 }, hh[2] = { 0,0 }, pw[2] = { 1, 1 };
19
20      for (int i = 0; i < M; i++) {
21          for (int j = 0; j < 2; j++) {
22              h[j] = (h[j] * p + pat[i]) % MOD[j];
23              hh[j] = (hh[j] * p + s[i]) % MOD[j];
24              if (i) pw[j] = (pw[j] * p) % MOD[j];
25          }
26      }
27
28      if (h[0] == hh[0] && h[1] == hh[1]) ans.push_back(1);
29      for (int i = 1; i + M - 1 < N; i++) {
30          for (int j = 0; j < 2; j++) {
31              hh[j] = (p * (hh[j] - pw[j] * s[i - 1]) + s[i + M - 1]) % MOD[j];
32              if (hh[j] < 0) hh[j] += MOD[j];
33          }
34          if (h[0] == hh[0] && h[1] == hh[1]) ans.push_back(i + 1);
35      }
36
37      printf("%d\n", (int)ans.size());
38      for (int&it : ans) printf("%d ", it);
39
40      return 0;
41  }
```

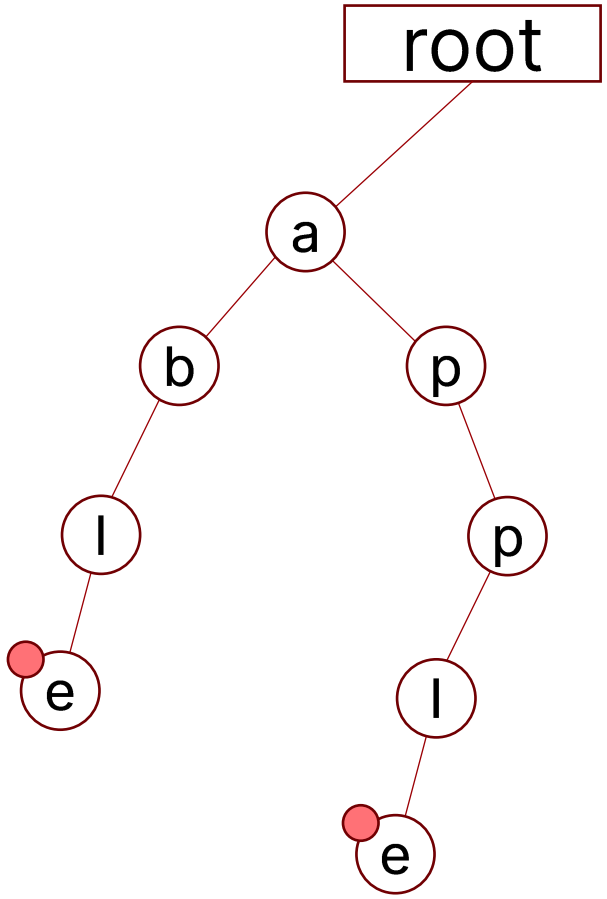
#28~34 double hash 두개에 대해
각각 대응하여야 매칭 성공

Trie

- Basic Concept
- Binary Trie

- 대표적인 탐색형 자료구조
  ⇒Binary Search Tree $O(\log n)$
- 길이 M인 문자열의 경우: $O(M \log n)$

- 메모리 소모up, but $O(M)$에 문자열 탐색이 가능한 자료구조



Basic Concept

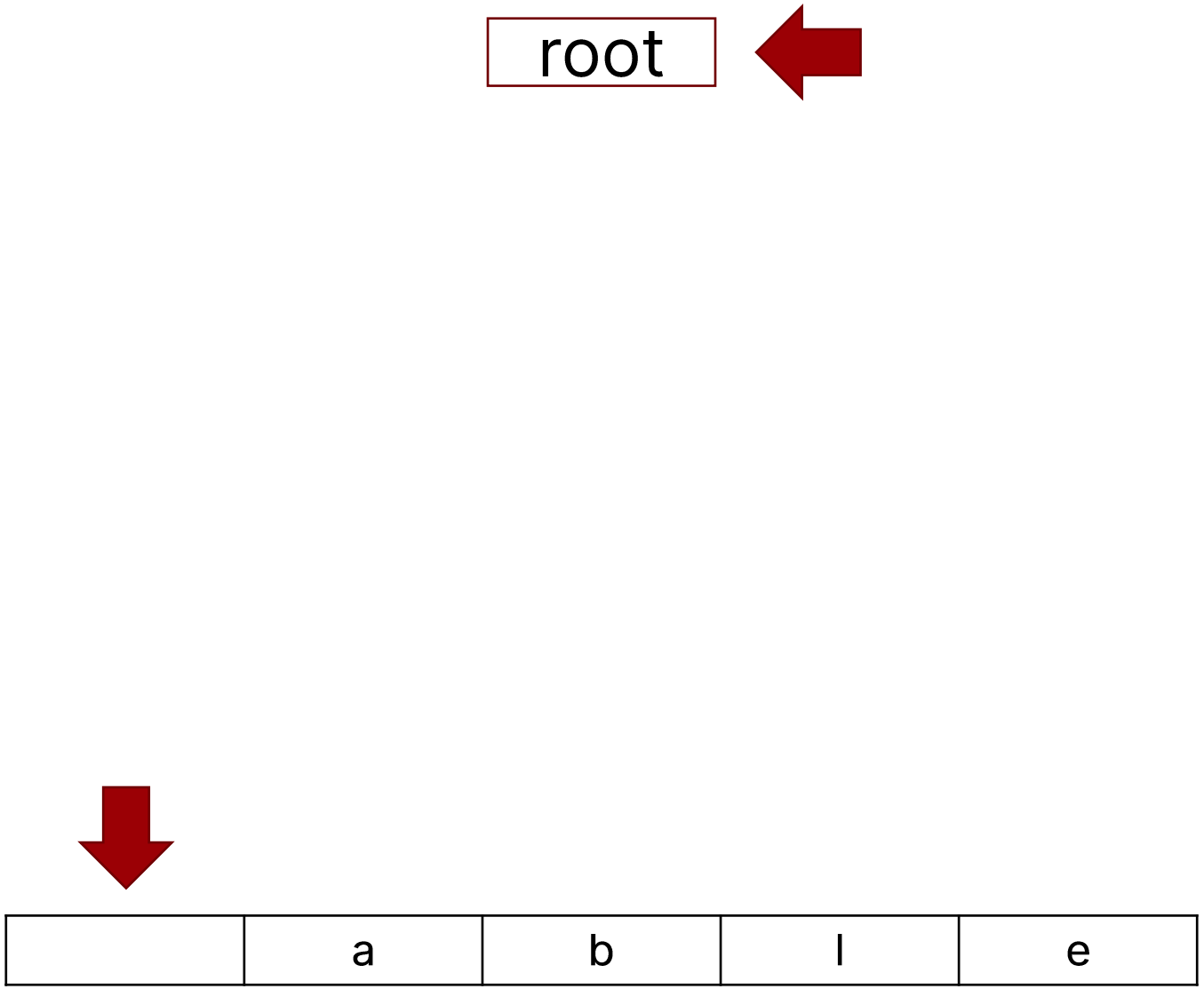
- 트리
- 각 node마다 한 문자(혹은 string), 다음 문자로 가기 위한 link 포함
- 부모 -> 자식 간 연결을 통해 다음 문자로 이동
- 단어 뿐만 아니라 접두사 또한 저장 가능
- Kind of DFA(Deterministic Finite Automata)
- Total memory: node크기 × 등장하는 문자열의 길이 합

- Why no hashing?

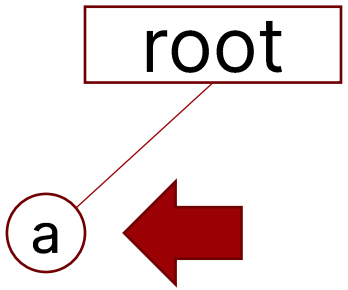
  ⇒해싱의 경우 alphabetical order과 같은 성질이 깨지게 됩니다.

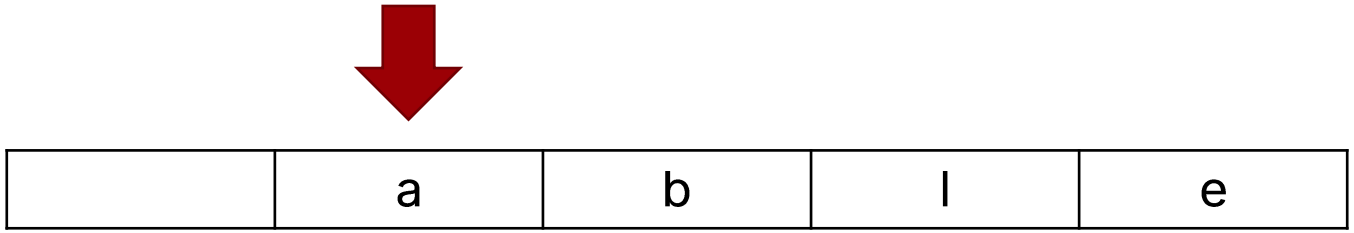
- node의 기본 구성
  - lsterminal: 저장한 문자열의 끝인지, prefix인지 구분
  - child: 다음 노드로 이동하기 위한 pointer

- Trie insertion

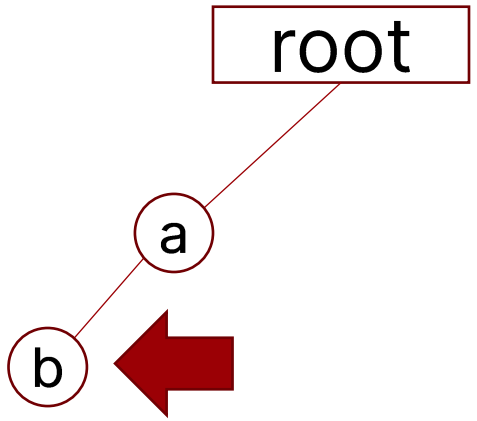


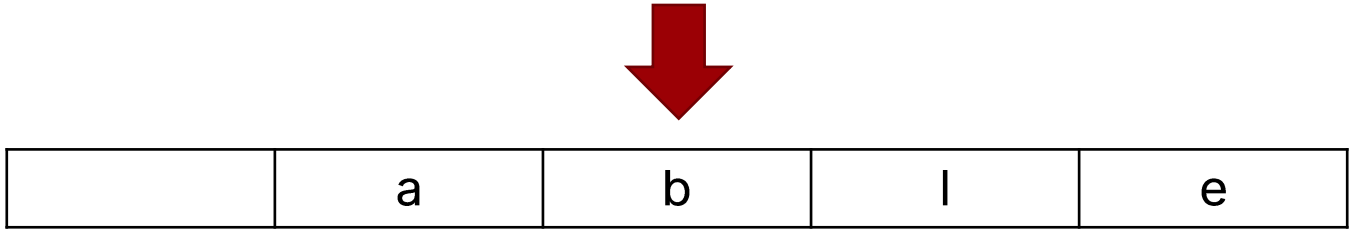
- Trie insertion



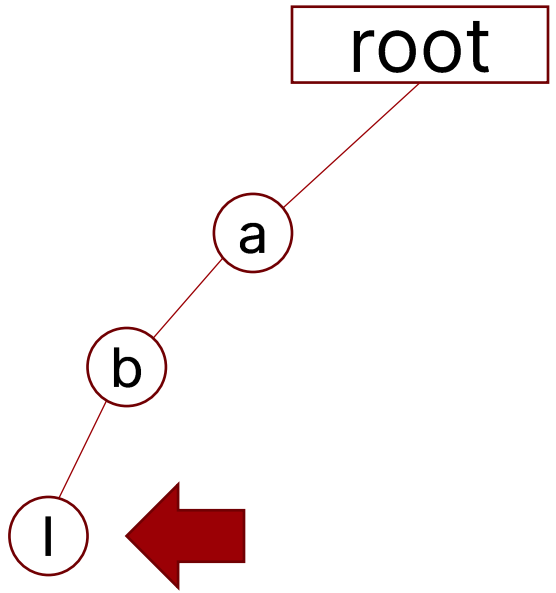


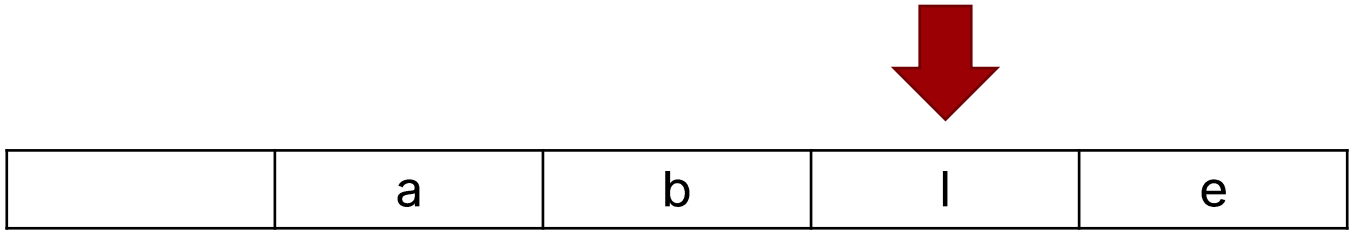
- Trie insertion



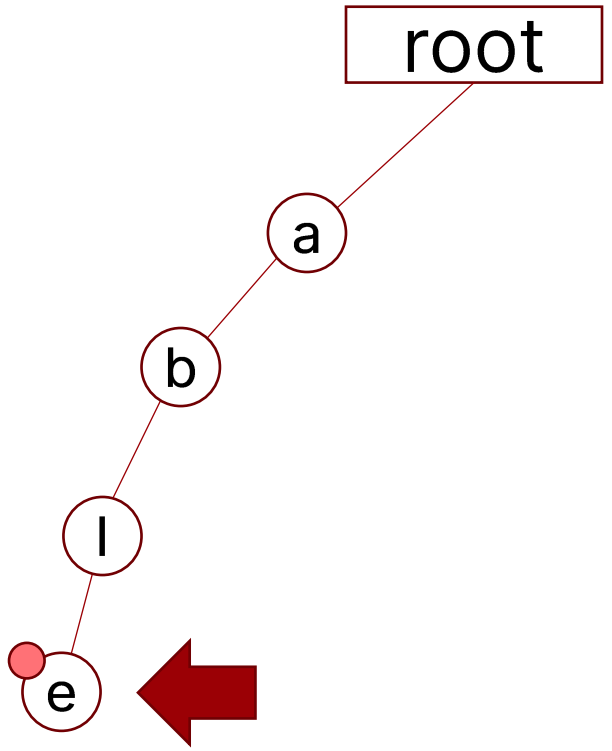


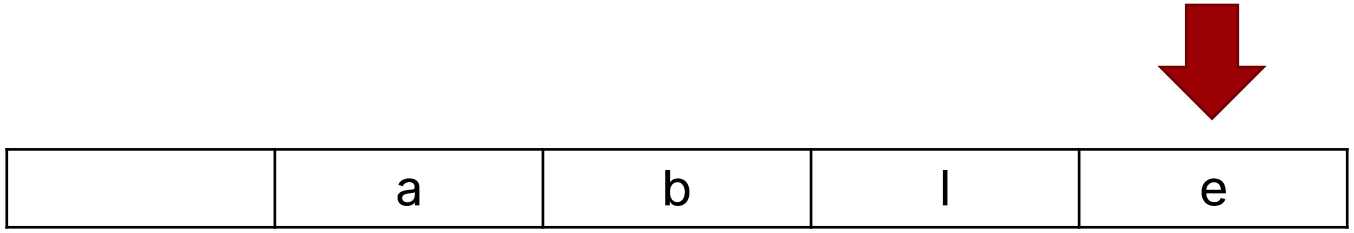
- Trie insertion



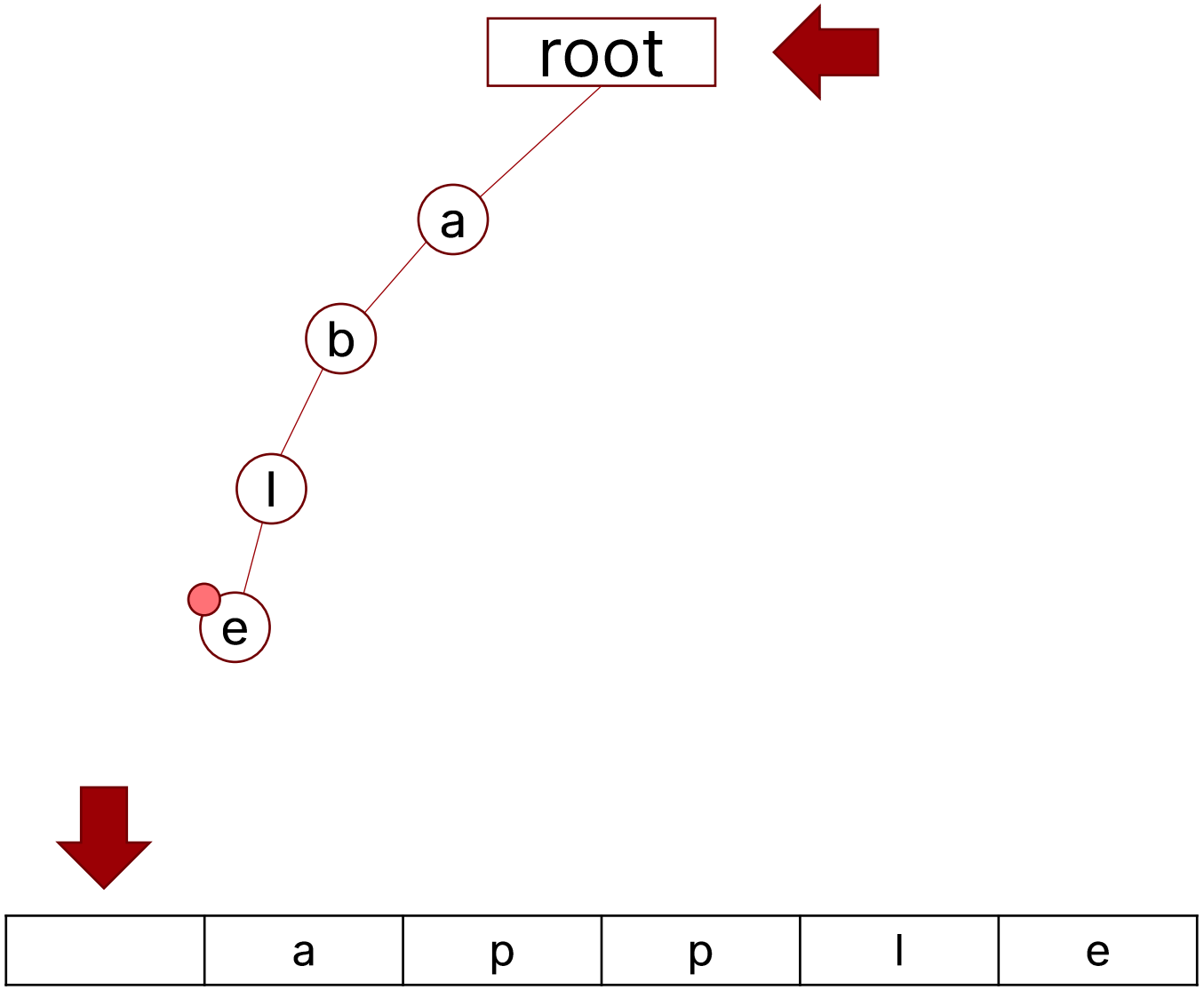


- Trie insertion

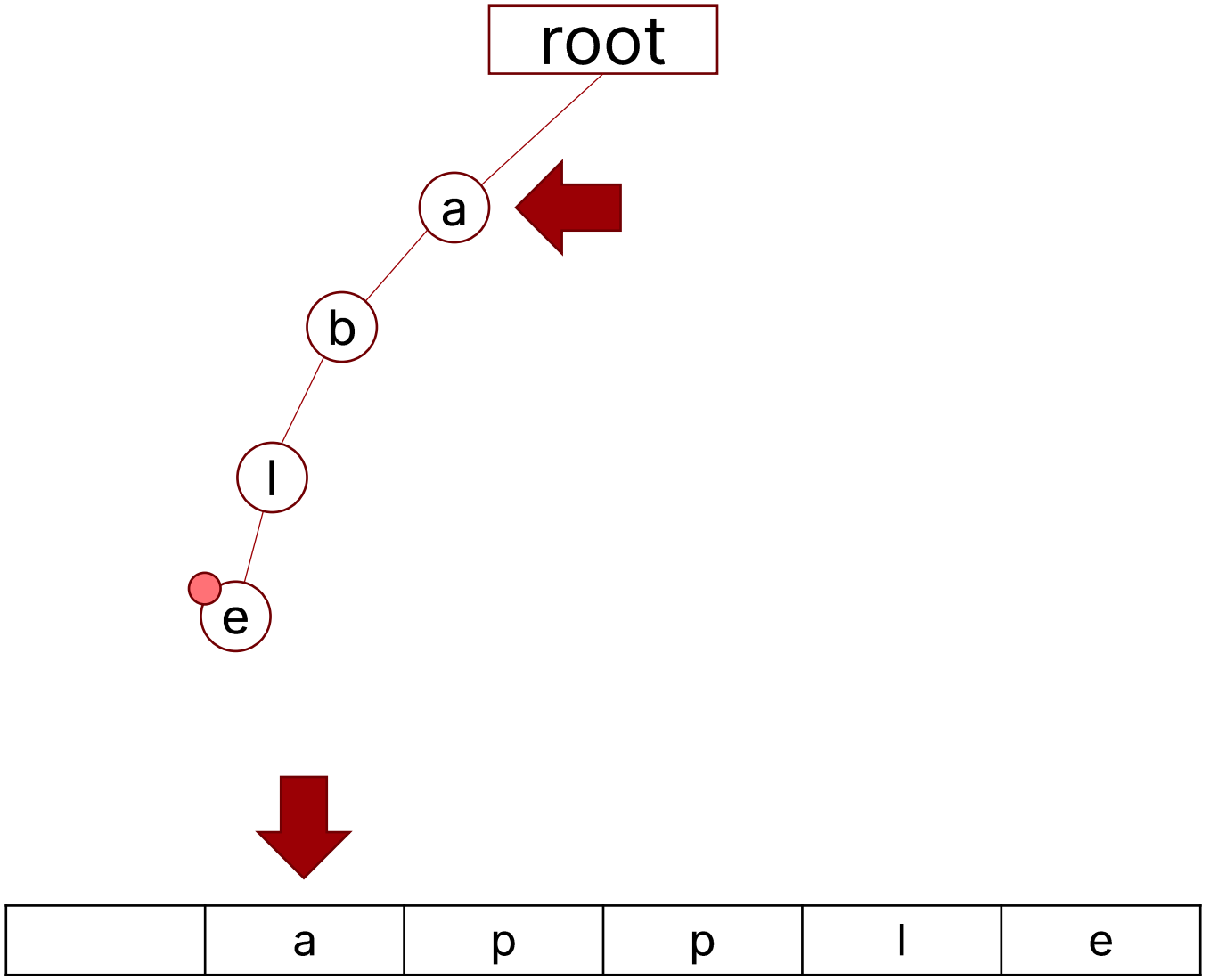




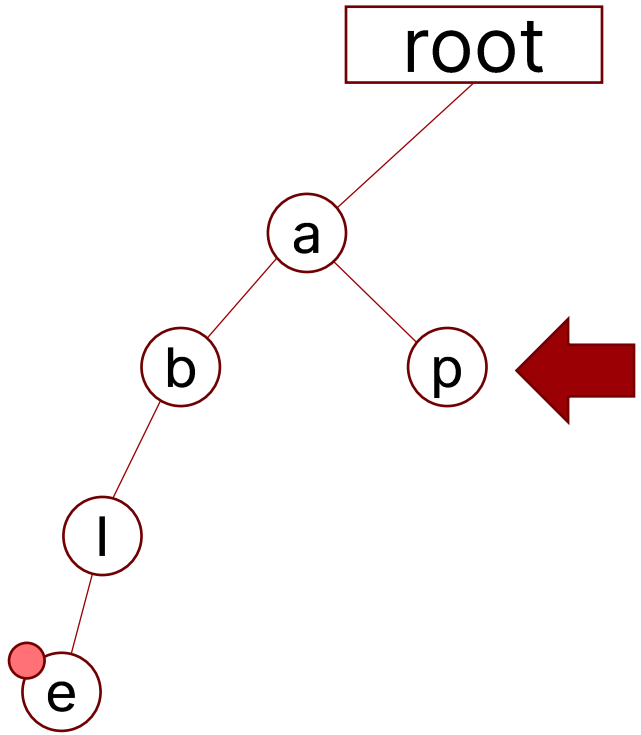
- Trie insertion



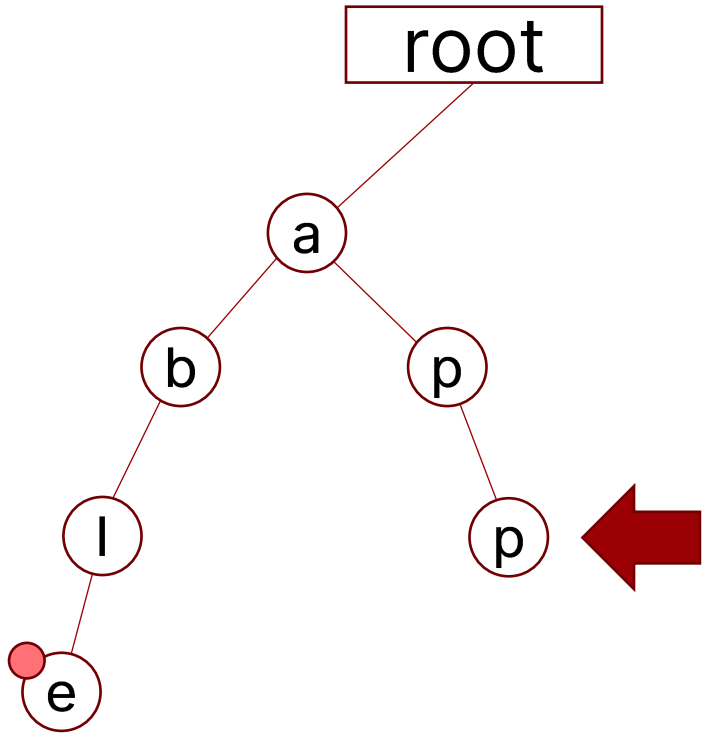
- Trie insertion

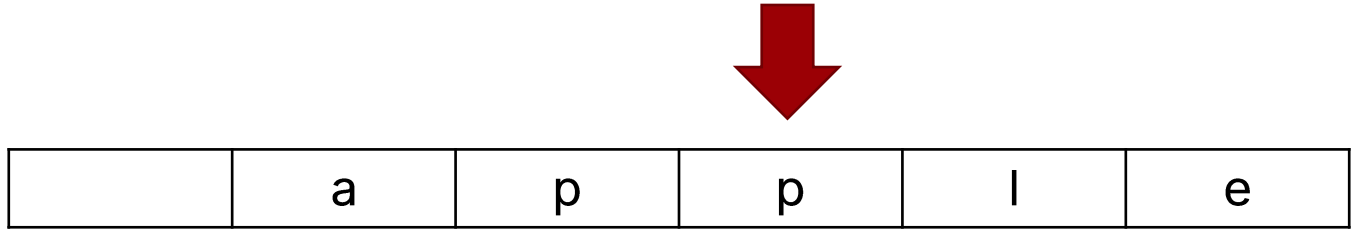


- Trie insertion

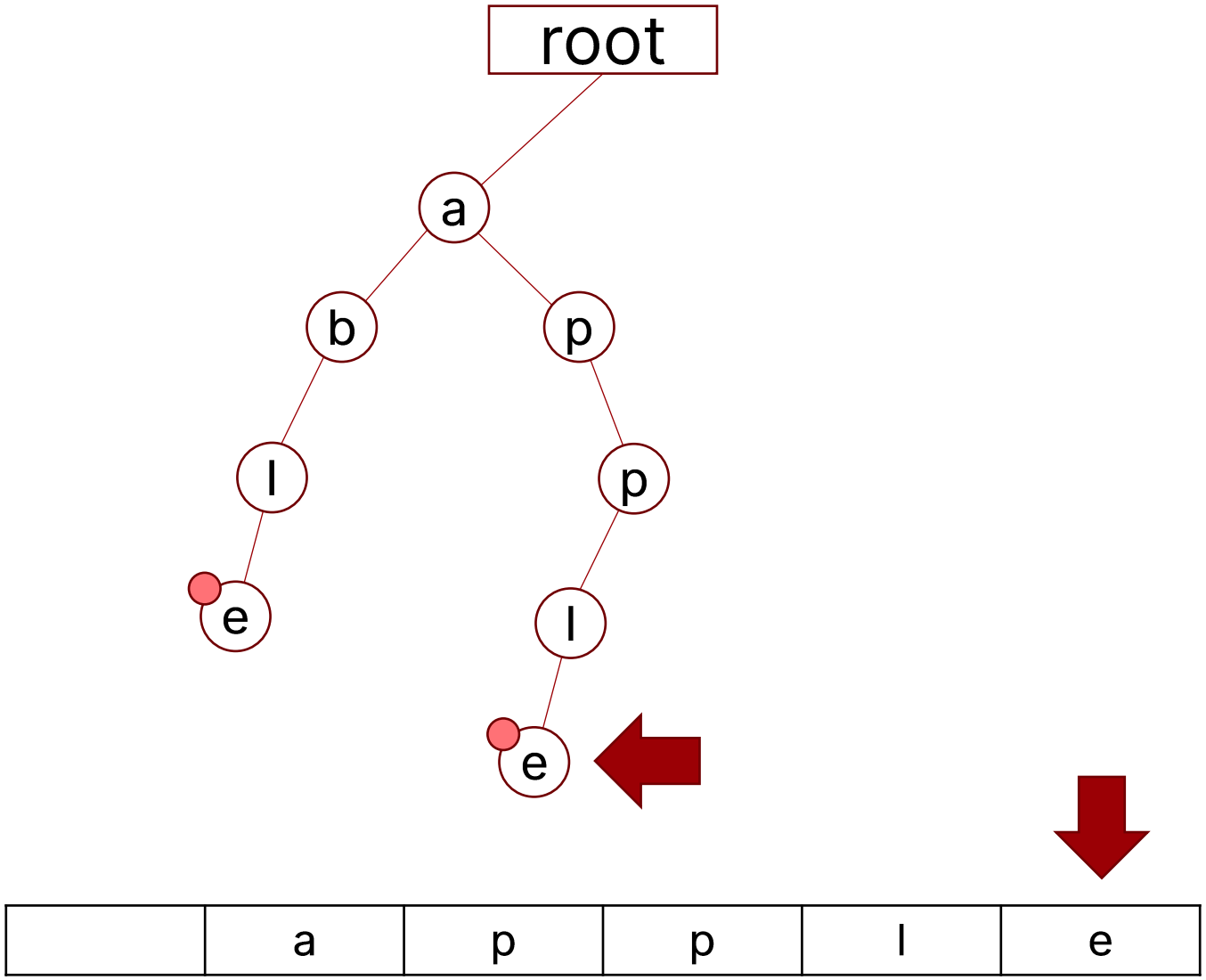


- Trie insertion

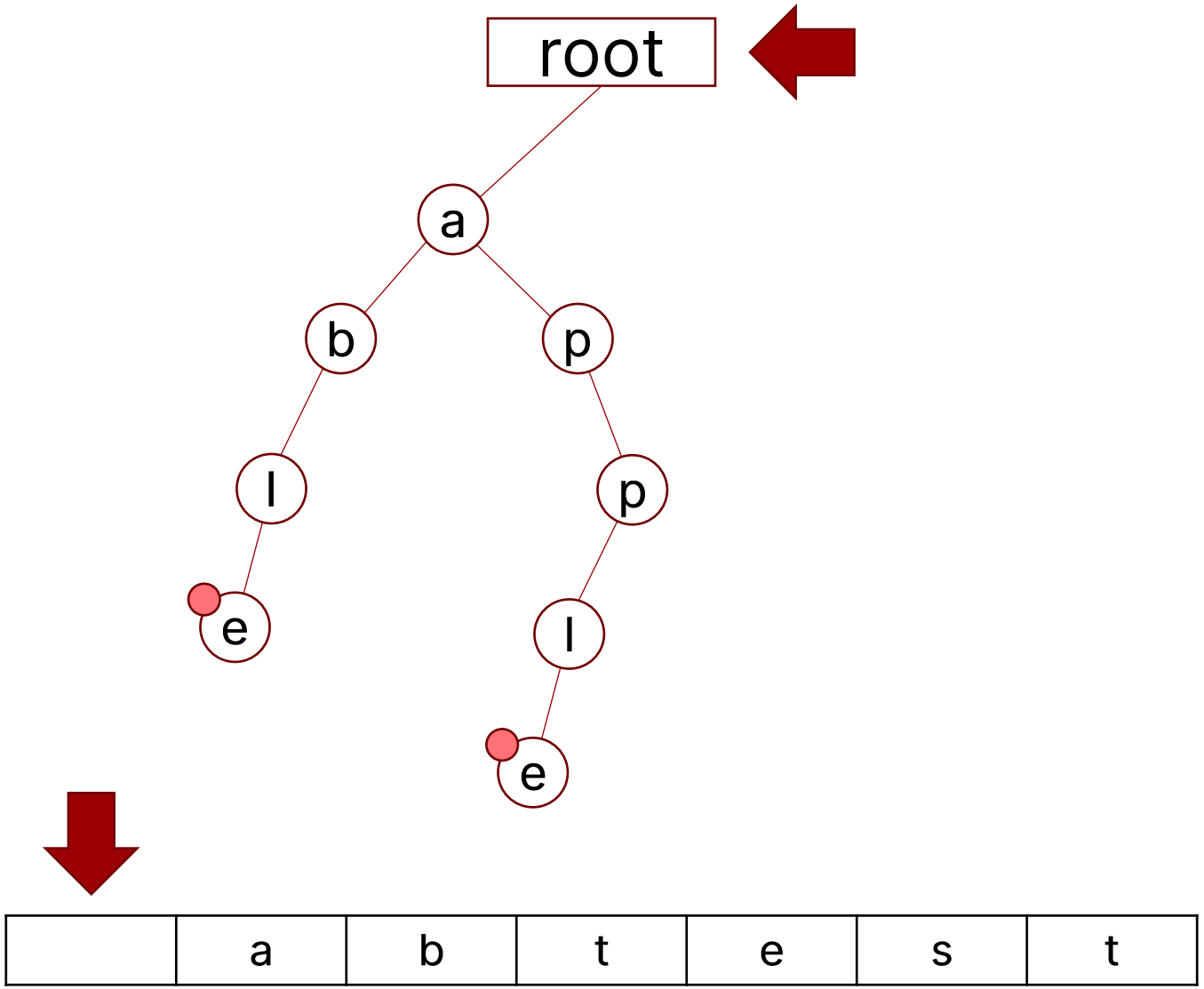




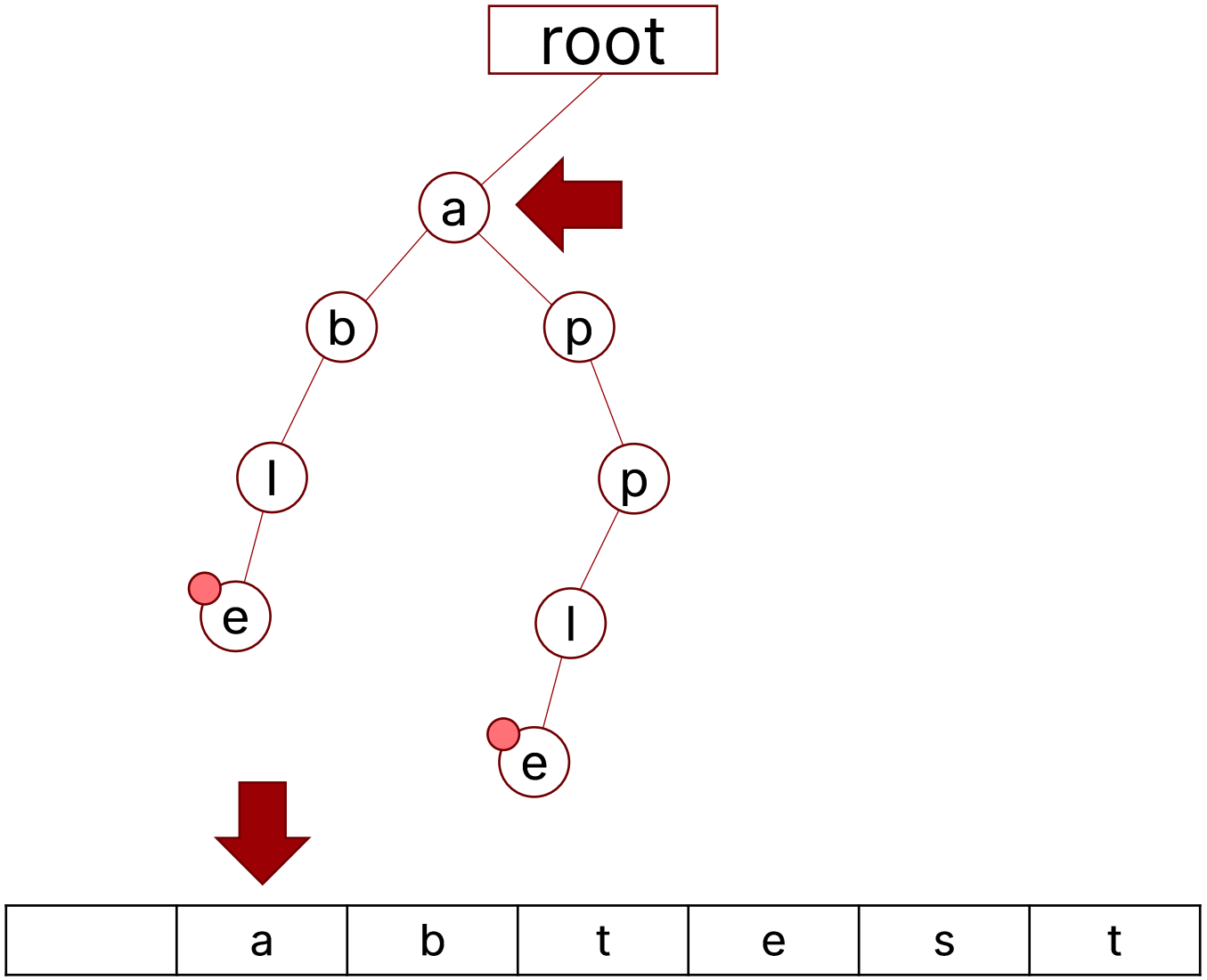
- Trie insertion



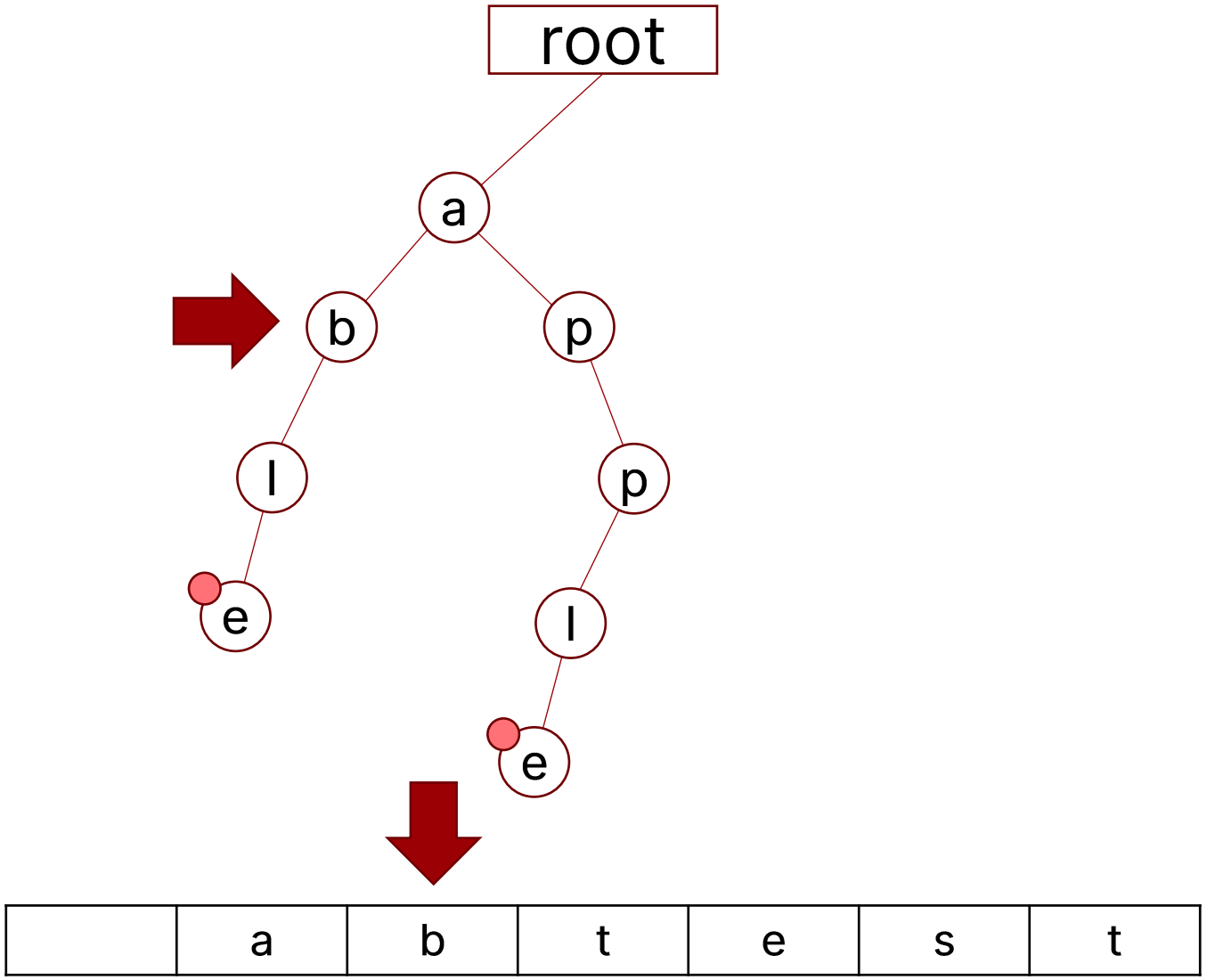
- Search



- Search

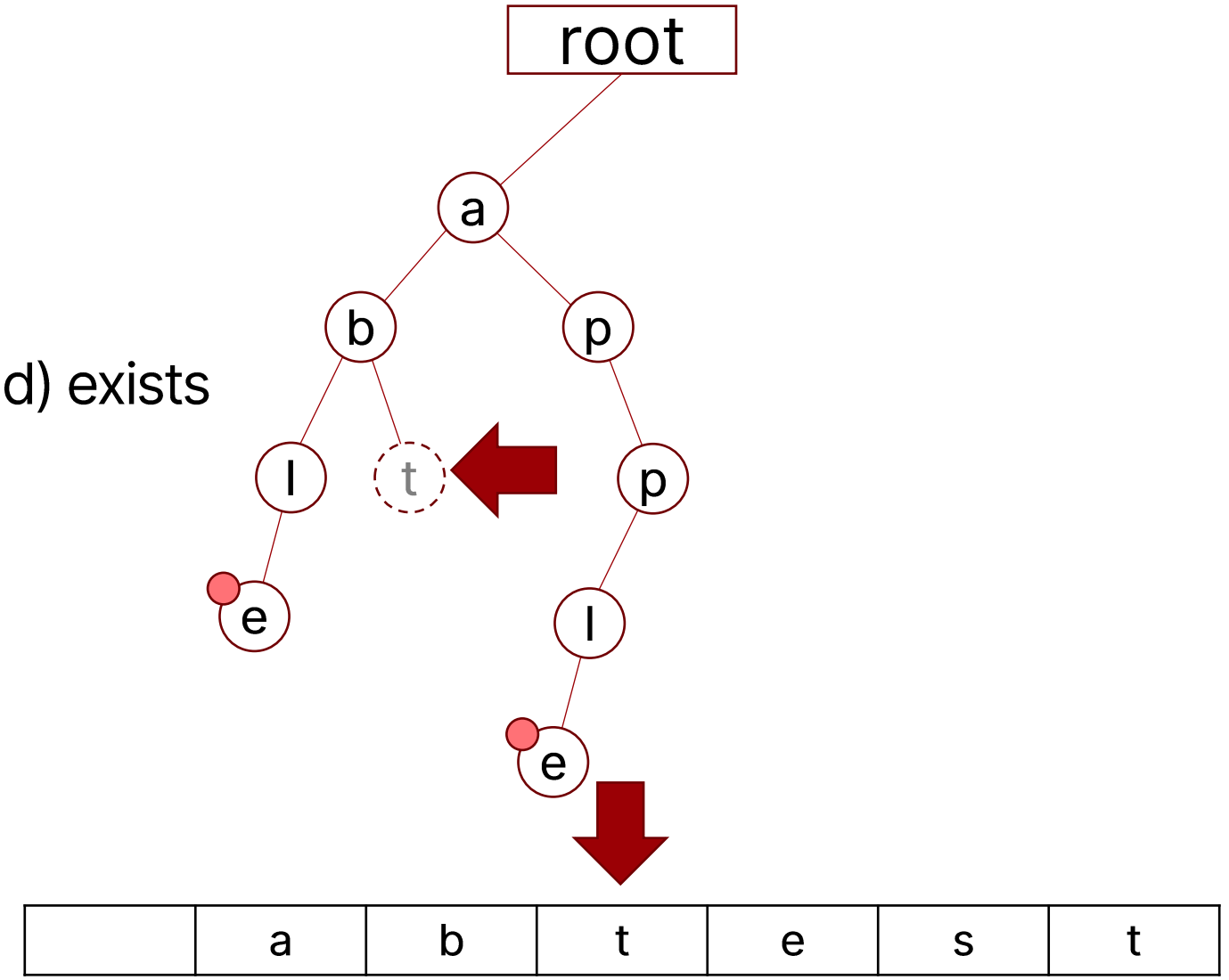


- Search

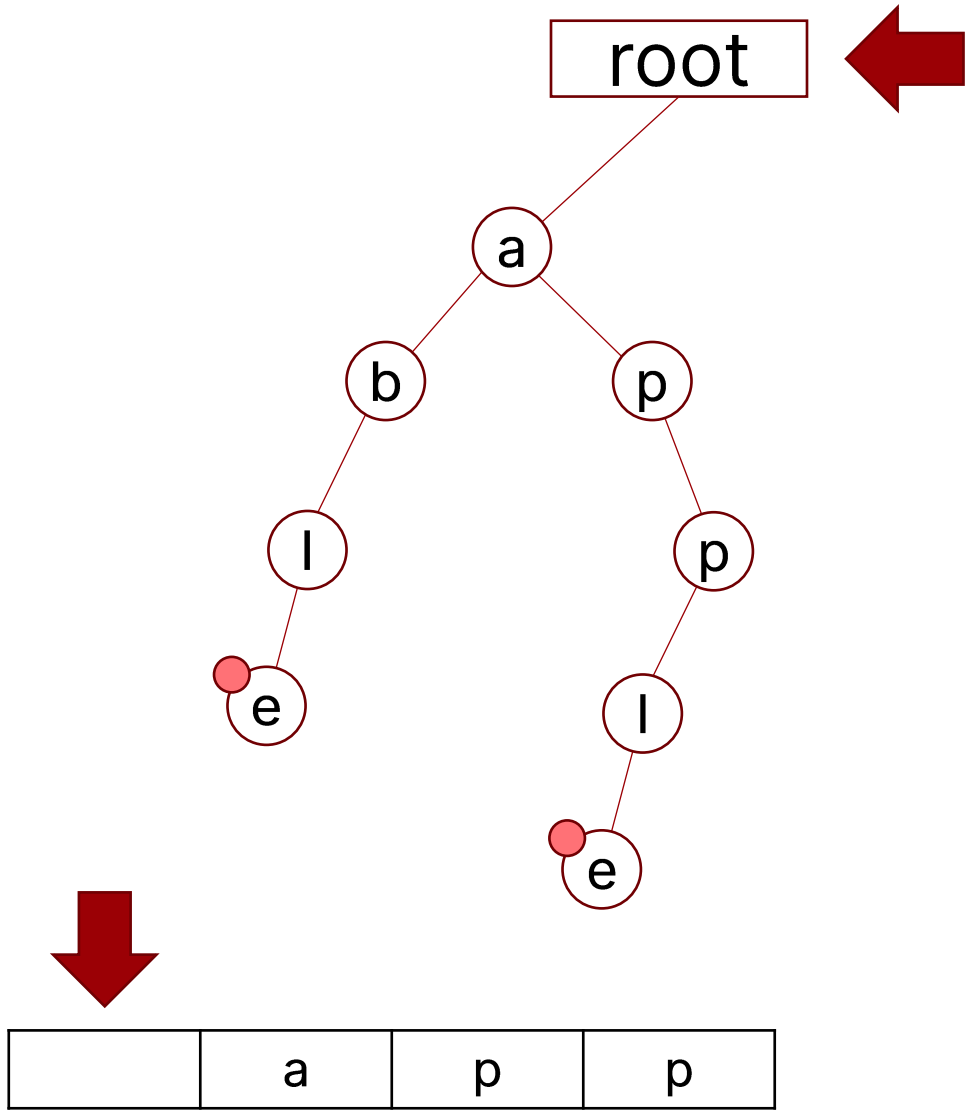


- Search

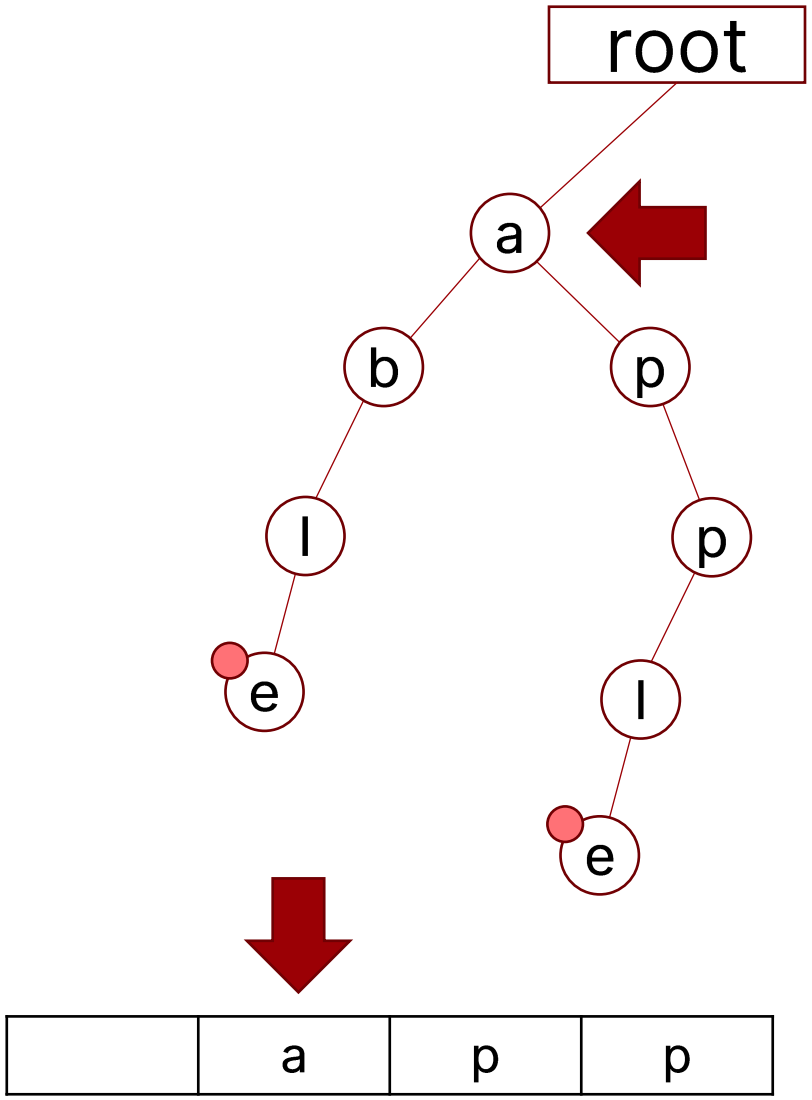
No path(child) exists



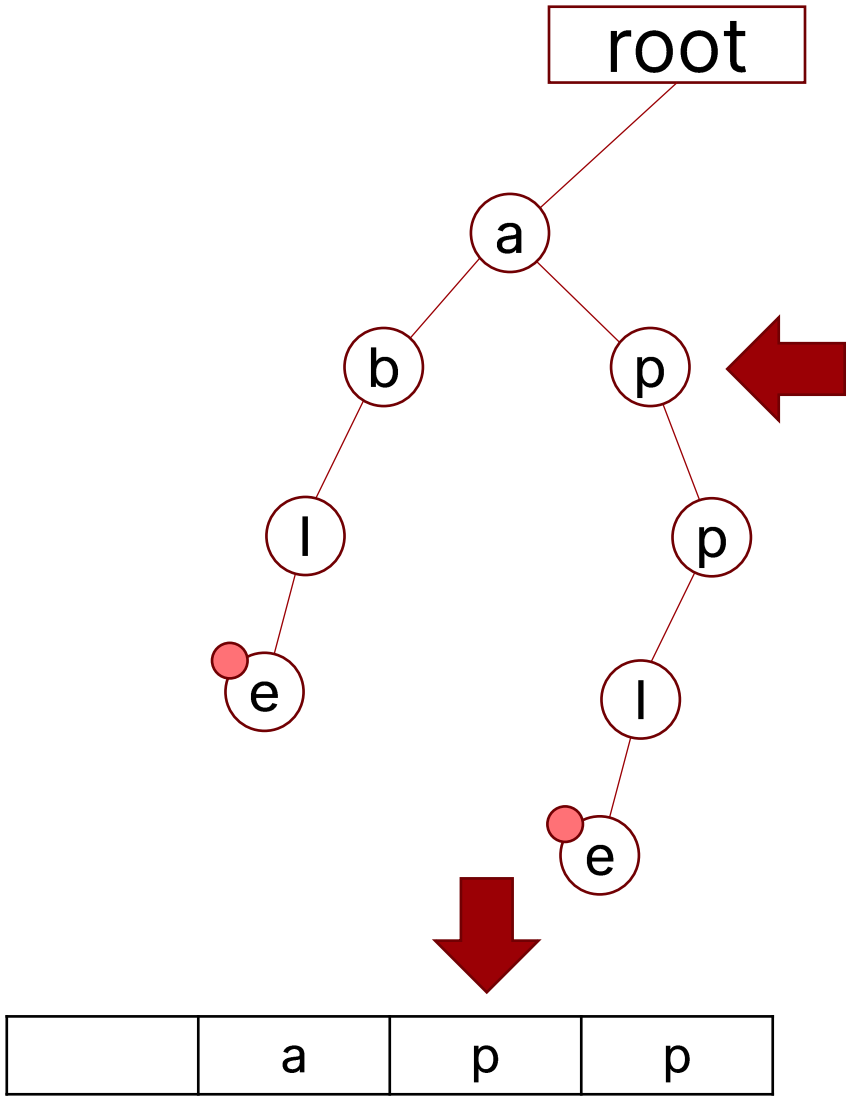
- Search



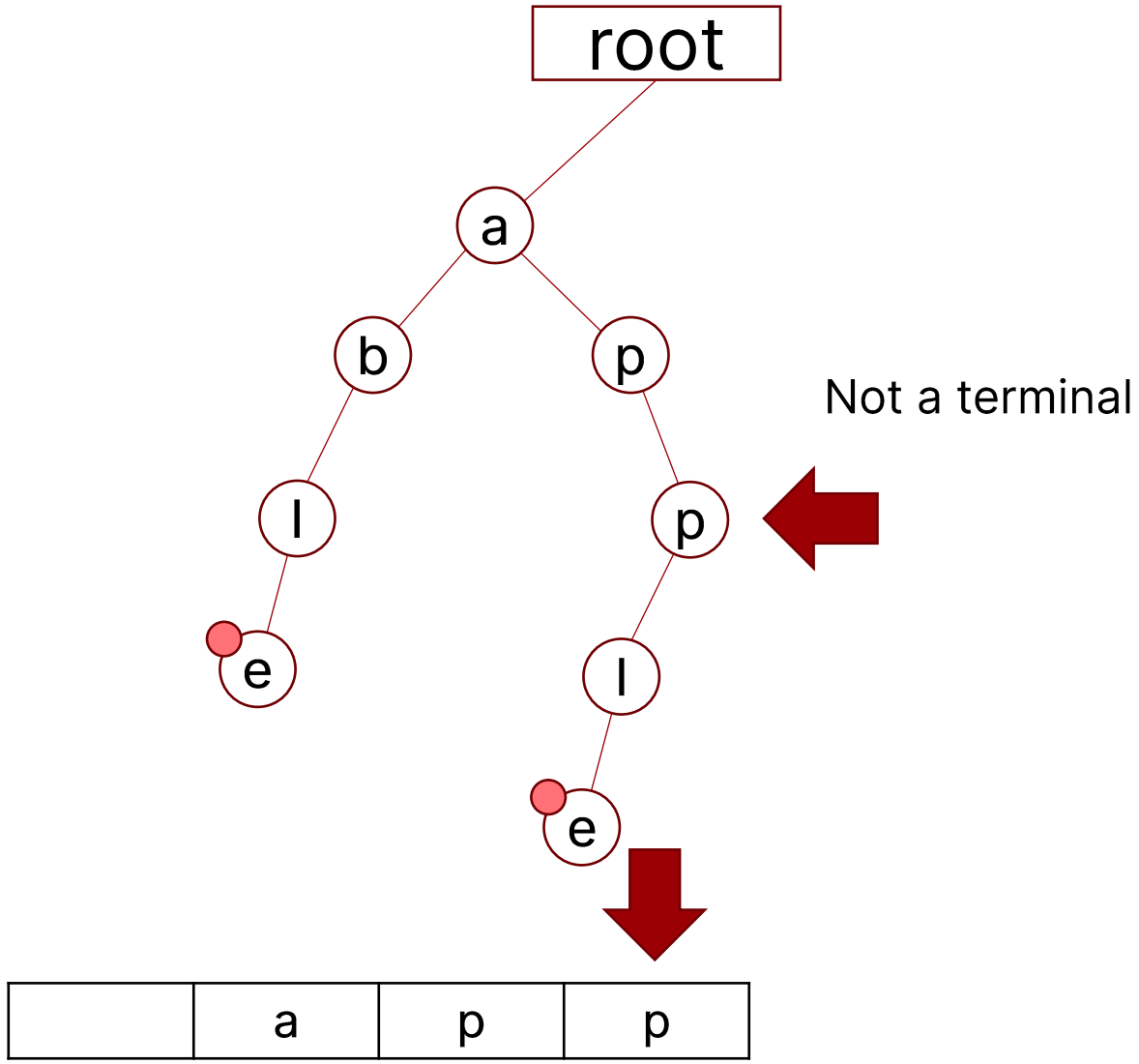
- Search



- Search



- Search



- Construction

1) 길이 없으면 만들어라

2) 길이 있으면 있는 곳으로 가라

3) 끝났으면 terminal 표시 해주자.

- Construction with array, without pointer

1) 길이 없으면 만들어라

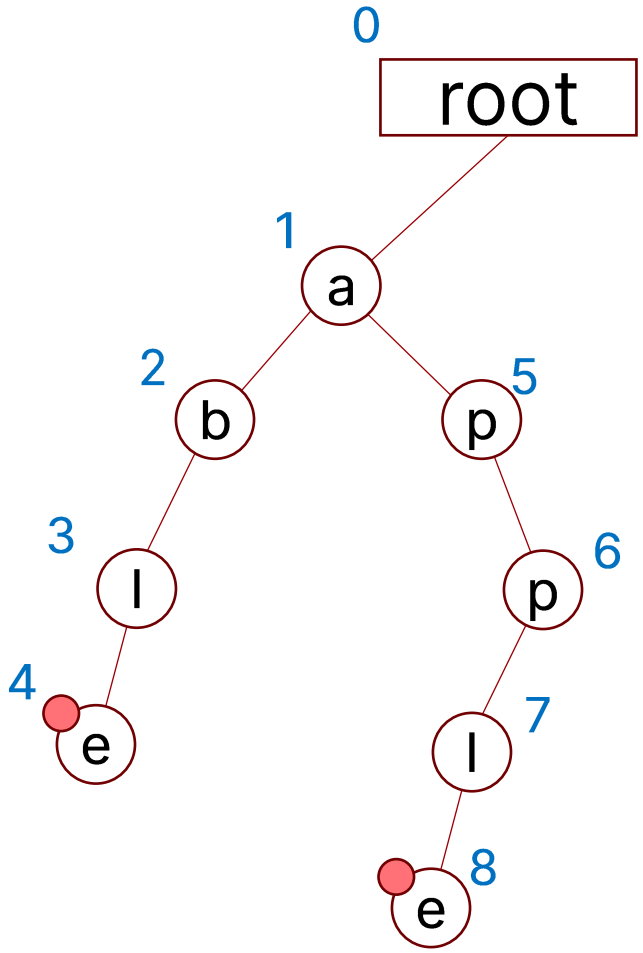
total index += 1

2) 길이 있으면 있는 곳으로 가라

move to existing index

3) 끝났으면 terminal 표시 해주자.

IsTerminal[index] = True;



Implementation

Problem 1

- 4×4 board에서 단어장에 존재하는 단어 찾기 게임
- 단어의 수 $w(1 < w < 300{,}000)$, 길이 $l(0 < l < 8)$
- 보드의 개수 $b(1 < b < 30)$, 대문자로만 구성
- board 내 이동은 인접한 가로, 세로, 대각선으로만 가능 (8방향)
- 단어 길이당 배점
  - 1,2 : 0점
  - 3,4 : 1점
  - 5 : 2점
  - 6 : 3점
  - 7 : 5점
  - 8 : 11점
- 얻을 수 있는 최대 점수, 가장 긴 단어, 찾은 단어의 수를 구해보자.
- 시간 제한: 10초

Word list
- ICPC
- ACM
- CONTEST
- GCPC
- PROGRAM

A	C	M	A
A	P	C	A
T	O	G	I
N	E	S	T

Problem 1

- 모든 경우의 수는 몇 개일까?
  1. 보드의 수 : 30개
  2. 시작 지점 : 16개
  3. 한 점에서 나올 수 있는 단어의 개수 : $< 7^7 \ (= 823{,}543)$

$$\prod = 395{,}300{,}640$$

- 단어 길이의 기댓값 : 4.5, trie로 위의 모든 경우에 대해 탐색을 시도한다면?

$$1{,}778{,}852{,}880 \ (< 1.8 \times 10^9)$$

- 시간 제한: 10초

- 문제 해결 과정

1. Trie 생성, 단어 리스트 추가
2. 구해야 하는 정보 확인(점수, 가장 긴 단어, 찾은 단어의 수)
3. 모든 점에 대하여, 한점으로부터 시작하여 나올 수 있는 모든 케이스 추출(Backtracking)
4. 중복하여 찾은 경우 방지를 위한 설계 필요

Binary Trie

- Branch가 0 또는 1만 존재하는 trie
- integer의 저장 또는 탐색을 상수시간(bit 개수)에 처리 가능

Problem 2

- $N(2 \leq N \leq 100{,}000)$개의 수 $A_1, A_2, \ldots, A_N (0 \leq A_i \leq 10^9)$
- 임의의 두 수를 xor하여 얻을 수 있는 최대값

Problem 2

- 두 수의 xor연산

1	0	1	1	0	0	1	1	0	1
0	1	1	1	1	0	0	1	1	0
1	1	0	0	1	0	1	0	1	1

- 하위 비트들의 합 < 상위 비트 1개의 값 (super-increasing subsequence)

Problem 2

- 문제 해결 과정

1. 주어진 수들을 binary trie에 추가
2. 각각의 수들에 대하여, 상위 비트가 최대한 다르도록 경로 추적(each $O(1)$)
   1) 현재 비트와 다른 비트로 가는 경로가 있을 경우, 해당 비트에 대응하는 값만큼 +
   2) 경로가 없는 경우 값 추가 없이 탐색
3. 구한 각 값들 중 최대값 추출

QnA