Trie Implementation

Q - 208

Q - 211

Q - 1804

Advantages & Disadvantages of Trie over Hashing.

Applications of Tries →
- Autocomplete System
- Word Suggestion System
- Spell checker / Autocorrect
- Prefix Matching
- Searching (Phone directory, dictionary

Trie or prefix tree is a tree data structure used to efficiently store and retrieve keys in a data set of strings.

① Implement Trie class

a) Trie() initializes the trie object

b) void insert (String word) inserts string word into trie.

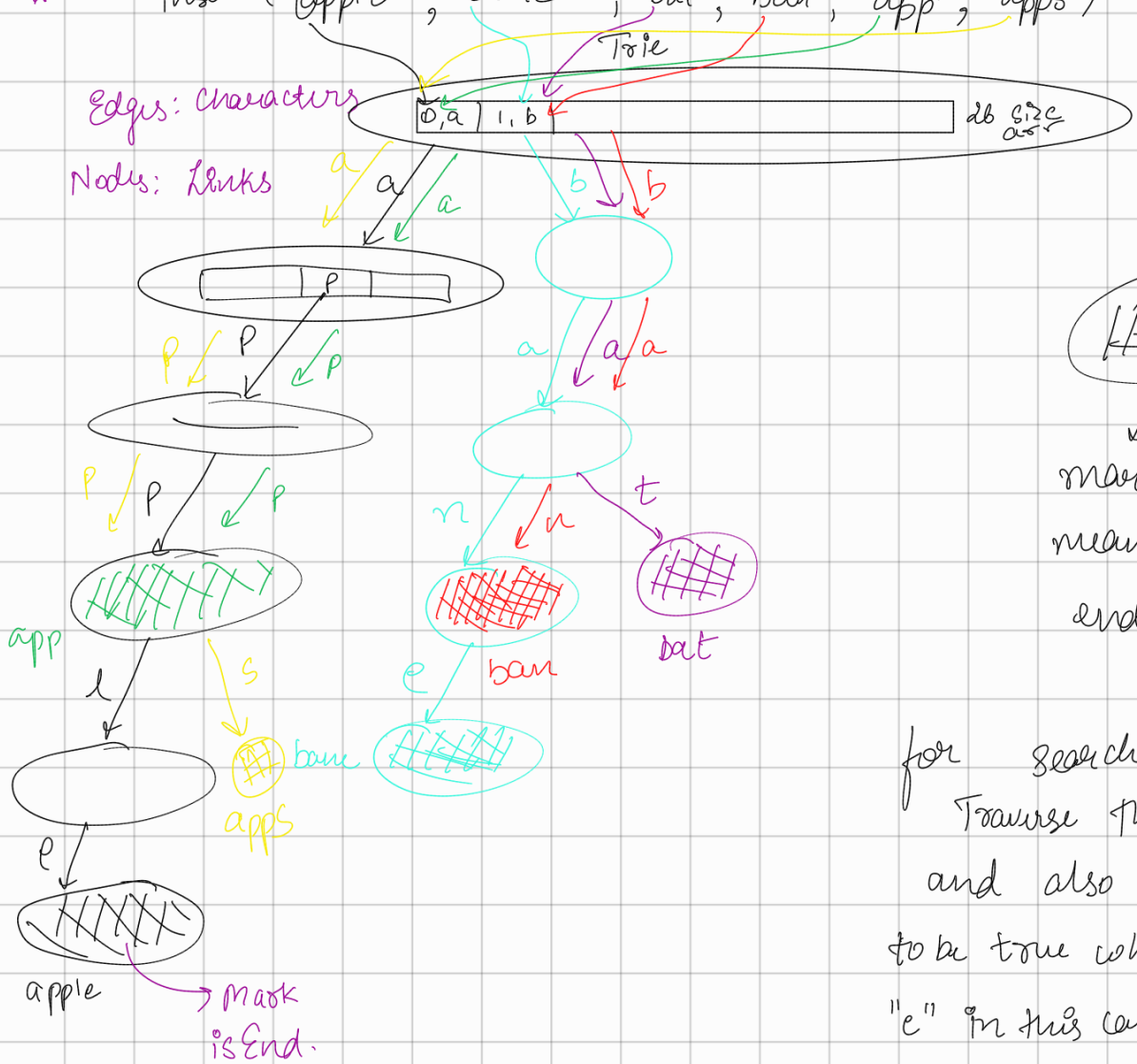
c) boolean search (String word) , checks if word is present in trie

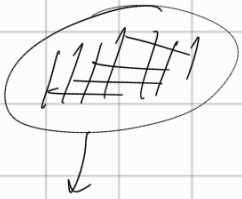
d) boolean startsWith (String prefix) → true if pow inserted words has prefix & false otherwise

Hashset	vs	Trie
↓		↓
String		O(Length)
O(Length)		find
		delete
		insert

Prefix O(N×L) ← set of string is huge → Collisions are increased — O(L)

☆ insert("apple", "bane", "bat", "ban", "app", "apps")





marks isEnd means a word ends here

for search("apple")
Traverse the trie again and also check for isEnd to be true when string ends at "e" in this case.

If all characters are found & isEnd exists then word is in trie

Nodes do not represent character, but links do.

search("bank") ⟶ b ✓, a ✓, n ✓, k ✗ → false

search("ap") ⟶ a ✓, p ✓, isEnd at p ✗ → false

☆ To check prefix, same as search, just don't check isEnd.

prefix's of apple are : apple, appl, app, ap, a

prefix("ban") → b ✓, a ✓, n ✓, DO NOT CHECK IS END, ✓ true

prefix ("bank") → b✓, a✓, n✓, k✗ ✗ false

CODE:

```
class Trie {
    public static class Node {
        private Node[] children = new Node[26];
        private boolean isEnd = false

        public boolean contains (char ch) {
            return (children[ch-'a'] != null);
        }

        public Node get (char ch) {
            return children[ch-'a'];
        }

        public void set (char ch) {
            children[ch-'a'] = new Node ();
        }

        public boolean getEnd() {
            return isEnd;
        }

        public void setEnd() {
            isEnd = true;
        }
    }

    Node root;
```

```
public Trie() {
    root = new Node();
}

public void insert(String word){
    Node curr = root;
    for(int i=0; i< word.length; i++){
        char ch = word.charAt(i);
        if(curr.contains(ch) == false)
            curr.set(ch);
        curr = curr.get(ch);
    }
    curr.SetEnd();
}

public boolean search(String word){
    Node curr = root;
    for(int i=0; i< word.length; i++){
        char ch = word.charAt(i);
        if(curr.contains(ch)== false)
            return false;
        curr = curr.get(ch);
    }
    return curr.getEnd();
}

public boolean startsWith(String prefix){
    Node curr=root;
    for(int i=0; i< prefix.length(); i++){
        char ch = prefix.charAt(i);
```

```
            if (curr.contains(ch) == false)
                return false;
            curr = curr.get(ch);
        }
        return true;
    }
}
```

* Search("b.c") → false

. means any character can be counted

Search("a.p") → true where . can be p.

So here res: Search(aap) or Search(abp) - - - Search(azp)

or means
if any one is true
res is true

③ Print prefix, go to word end then apply DFS.
Every time print when isEnd > 0

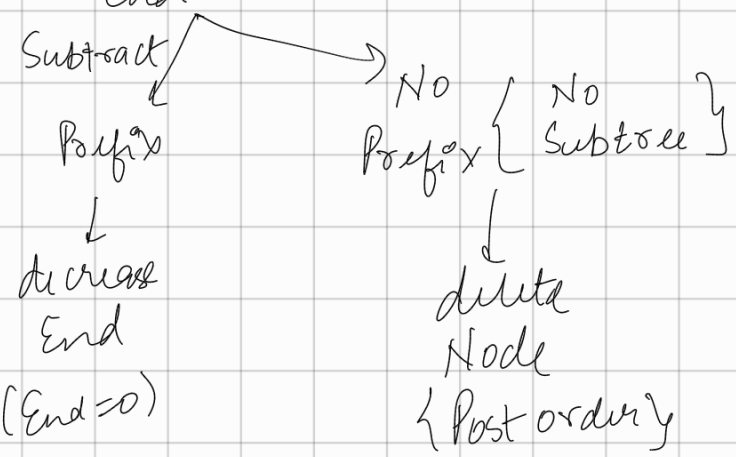
Use isEnd as integer instead of boolean

CountWords EqualTo: go to word end, return isEnd
CountWords Starting with: go to word end, apply DFS
return sum of all isEnds

delete ("ban")

End > 1 → decrease end

End = 1



Optimize, use another variable for each node that stores count of words of whose the current trie structure is prefix of

when making a new word → prefix of ++

" deleting / erasing word → prefix of --

CODE

```
public class Trie {
    public static class Node {
        private Node[] children = new Node[26];
        private int isEnd=0, prefixCount=0;

        public boolean contains(char ch) {
            return (children[ch-'a'] != null);
        }

        public Node get(char ch) {
            return children[ch-'a'];
        }
```

```
    public void set(char ch){
        children[ch-'a'] = new Node();
    }

    public int getFreq(){
        return isEnd;
    }

    public int getPref(){
        return prefixCount;
    }

    public void increaseFreq(){
        isEnd++;
    }

    public void decreaseFreq(){
        isEnd--;
    }

    public void increasePref(){
        prefixCount++;
    }

    public void decreasePref(){
        prefixCount--;
    }
}

Node root;
```

```
public Trie(){
    root= new Node();
}

public void insert(String word){
    Node curr= root;

    for(int i=0; i< word.length(); i++){
        curr.increasePref();
        char ch= word.charAt(i);
        if(curr.contains(ch) == false)
            curr.set(ch);
        curr= curr.get(ch);
    }
    curr.increasePref();
    curr.increaseFreq();
}

public int countWordsEqualTo(String word){
    Node curr= root;

    for(int i=0; i< word.length(); i++){
        char ch= word.charAt(i);
        if(curr.contains(ch)== false)
            return 0;
        curr= curr.get(ch);
    }
    return curr.getFreq();
}
```

```
public int countWordsStartingWith(String word){
    Node curr = root

    for(int i=0; i<word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return 0;

        curr = curr.get(ch);
    }
    return curr.getPref();
}

public void erase(String word){
    if(countWordsEqualTo(word)==0)
        return;

    Node curr = root;

    for(int i=0; i<word.length(); i++){
        curr.decreasePref();
        char ch = word.charAt(i);
        curr = curr.get(ch);
    }

    curr.decreasePref();
    curr.decreaseFreq();
}
}
```

④ Map Sum Pairs.

- Using DFS.
- Optimized.

CODE:

```
class MapSum {
    public static class Node {
        private Node[] children = new Node[26];
        private int val=0;
        private int prefix=0;

        public Node get(char ch){
            return children[ch-'a'];
        }

        public int getval(){
            return val;
        }

        public void add(char ch){
            children[ch-'a'] = new Node();
        }

        public void setVal(int val){
            this.val = val;
        }

        public boolean contains(char ch){
            return (children[ch-'a'] != null);
        }
    }

    Node root;
    public MapSum(){
        root = new Node();
    }
```

```
public int search (String word){
    Node curr = root;
    for(int i=0; i < word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return 0;

        curr = curr.get(ch);
    }
    return curr.getVal();
}

public void insert(String word, int val){
    int oldVal = search(word);

    Node curr = root;

    for(int i=0; i < word.length(); i++){
        curr.pref += (val - oldVal);

        char ch = word.charAt(i);
        if(curr.contains(ch) == false)
            curr.add(ch);

        curr = curr.get(ch);
    }
    curr.pref += (val - oldVal);
    curr.setVal(val);
}
```

```
public int sum(String word){
    Node curr = root;
    for(int i=0; i< word.length(); i++){
        char ch = word.charAt(i);

        if(curr.contains(ch) == false)
            return 0;

        curr = curr.get(ch);
    }
    return curr.pref;
  }
}
```