Trie Notes

What is a Trie?

A Trie, also known as a prefix tree, is a type of search tree used in computer science for storing a dynamic set or associative array where the keys are usually strings. Unlike binary search trees, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.

Structure of a Trie

A Trie consists of nodes, each of which represents a string. Each node has a set of child nodes, and each child node represents a character in the string. The root node represents the empty string. Each node also has a boolean flag indicating whether the node represents the end of a string.

```
const int SIZE = 26; // Assuming only English letters
struct TrieNode {
    TrieNode* children[SIZE]; // Array of child nodes
    bool isEndOfWord; // Flag to mark the end of a word
};
```

Operations on a Trie

1. Insertion

Inserting a string into a Trie involves creating a new node for each character in the string, and linking each node to its child nodes.

Example:

Suppose we want to insert the strings "cat", "car", and "cart" into a Trie.

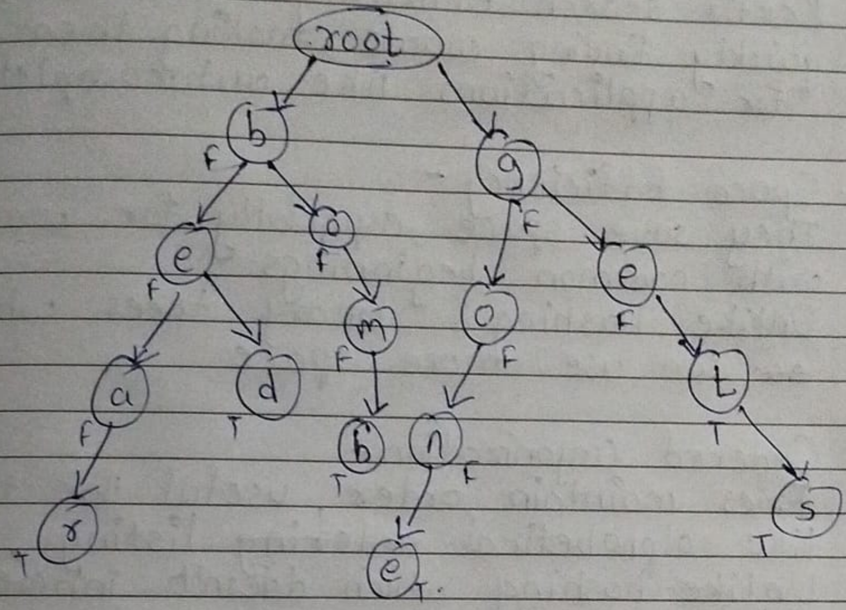
```
      root
     /   \
    c     null
   / \
  a   null
```

```
 / \
t    r
|     |
null  t
```

C++ Code for Insertion:

```
void insert(TrieNode* root, string key) {
    TrieNode* current = root;
    for (char c : key) {
        int index = c - 'a'; // Calculate the index for the
array
        if (!current->children[index]) {
            current->children[index] = createTrieNode();
        }
        current = current->children[index];
    }
    current->isEndOfWord = true;
}
```

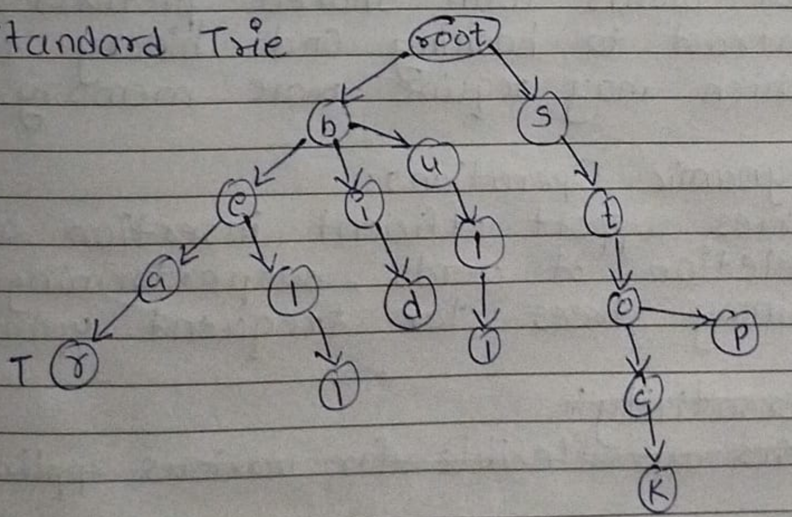
Q. Insert -
bear, bed, get, gone, bomb

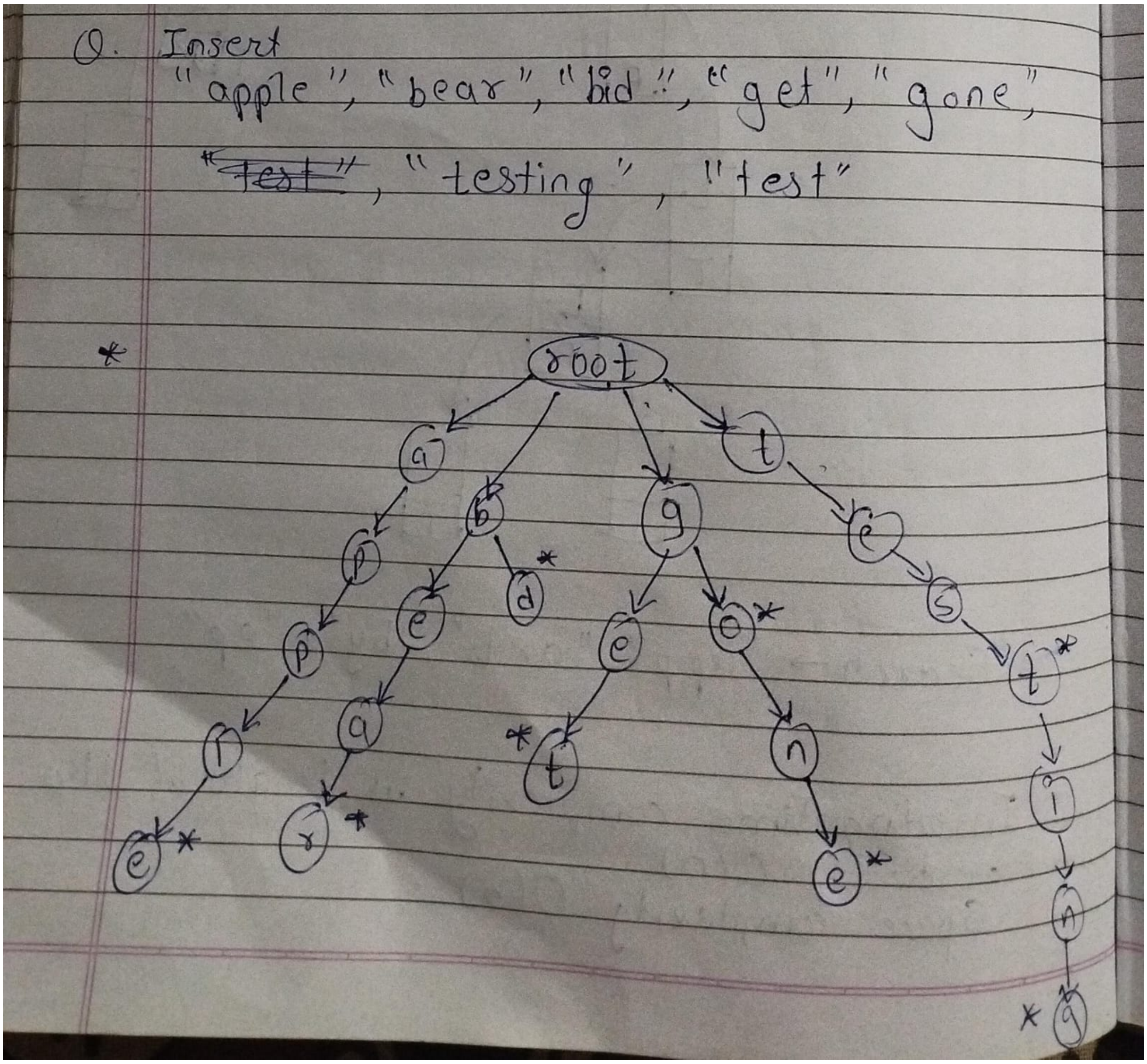


Search → "get", "gets"

Q. { bear, bell, bid, bull, stock, stop }

Standard Trie





2. Search

Searching for a string in a Trie involves traversing the Trie from the root node, following the child nodes corresponding to each character in the string.

C++ Code for Search:

```
bool search(TrieNode* root, string key) {
    TrieNode* current = root;
    for (char c : key) {
        int index = c - 'a';
        if (!current->children[index]) {
            return false;
        }
        current = current->children[index];
```

```
    }
    return current->isEndOfWord;
}
```

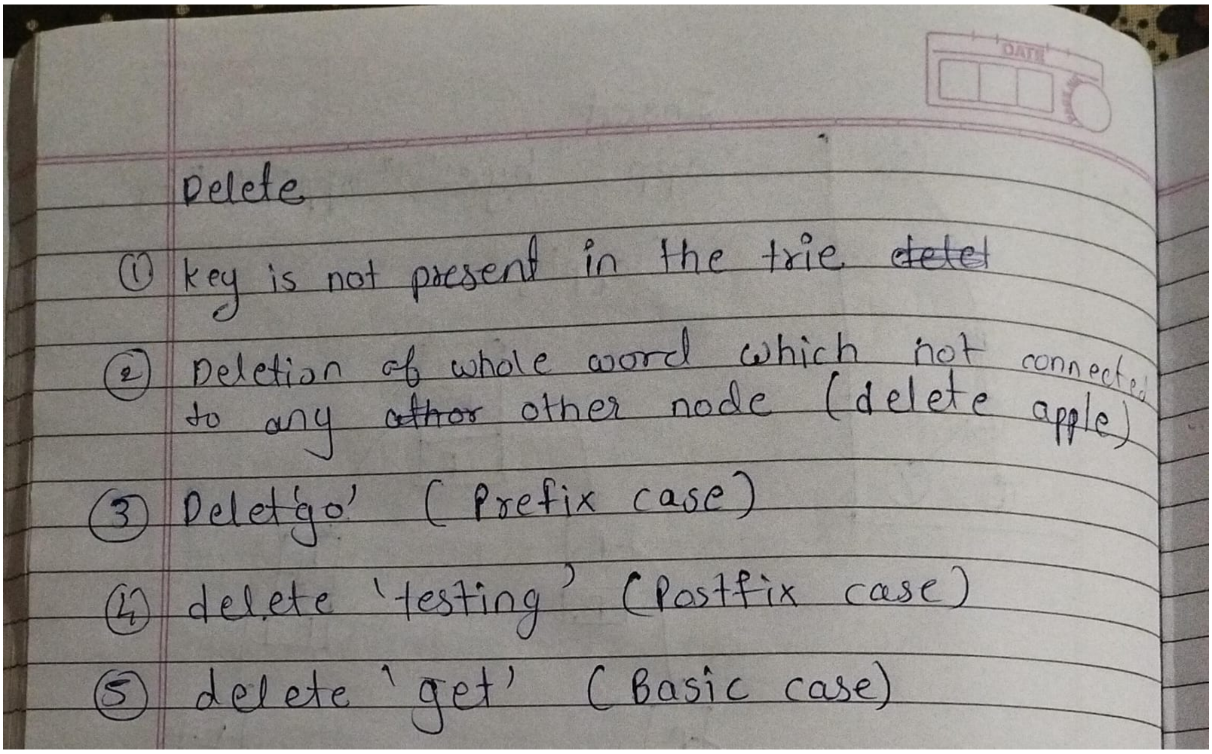
3. Deletion

Deleting a string from a Trie involves finding the node corresponding to the last character of the string, and marking it as not being the end of a word. If the node has no other children, it can be removed.

C++ Code for Deletion:

```
void deleteNode(TrieNode* node) {
    for (int i = 0; i < SIZE; i++) {
        if (node->children[i]) {
            deleteNode(node->children[i]);
        }
    }
    delete node;
}

void deleteKey(TrieNode* root, string key) {
    deleteKeyHelper(root, key, 0);
}

bool deleteKeyHelper(TrieNode* node, string key, int index)
{
    if (!node) {
        return false;
    }

    if (index == key.size()) {
        if (!node->isEndOfWord) {
            return false;
        }

        node->isEndOfWord = false;

        if (isEmpty(node)) {
```

```
            deleteNode(node);
            return true;
        }

        return false;
    }

    int charIndex = key[index] - 'a';
    if (!deleteKeyHelper(node->children[charIndex], key, index + 1)) {
        return false;
    }

    if (isEmpty(node) && !node->isEndOfWord) {
        deleteNode(node);
        return true;
    }

    return false;
}

bool isEmpty(TrieNode* node) {
    for (int i = 0; i < SIZE; i++) {
        if (node->children[i]) {
            return false;
        }
    }
    return true;
}
```



Delete

1. key is not present in the trie ~~detet~~
2. Deletion of whole word which not connected to any ~~othor~~ other node (delete apple)
3. Delet'go' (Prefix case)
4. delete 'testing' (Postfix case)
5. delete 'get' (Basic case)

4. Sorting

Sorting a Trie involves traversing the Trie in a depth-first manner, and printing out the strings in lexicographic order.

C++ Code for Sorting:

```
void dfs(TrieNode* node, string current, vector<string>& re
sult) {
    if (node->isEndOfWord) {
        result.push_back(current);
    }

    for (int i = 0; i < SIZE; i++) {
        if (node->children[i]) {
            dfs(node->children[i], current + (char)(i +
'a'), result);
        }
    }
}

vector<string> sortedKeys(TrieNode* root) {
    vector<string> result;
```

```
    dfs(root, "", result);
    sort(result.begin(), result.end());
    return result;
}
```

Applications of Trie

1. **Auto-complete**: Tries are used in auto-complete features of search engines and text editors to suggest possible completions of a partially typed string.
2. **Spell checking**: Tries can be used to implement spell checking algorithms that suggest corrections for misspelled words.
3. **Validating IP addresses**: Tries can be used to validate IP addresses by checking if a given IP address is valid or not.
4. **Data compression**: Tries can be used to compress data by storing a set of strings in a compact form.

Advantages of Trie

* Advantages

1. Prefix search efficiency : Tries excel at quickly finding words, making them ideal for applications like autocomplete

2. Space Efficiency -
They save space, especially for words with common beginnings, Unlike hashing & binary trees which ~~oa~~ can use more space

3. Ordered Organization -
Tries maintain order, useful for tasks like alphabetical ~~ordering~~ listing, Unlike hashing which doesn't inherently offer order

4. Memory Conservation :
They consume less memory, particularly for words with shared prefixes, in contrast to hashing and binary trees which may require more memory.

5. Dynamic Operations:
Tries support efficient insertion and deletion of words, outperforming binary trees for frequent updates.

Disadvantages of Trie

1. **Complexity**: Tries can be complex to implement and manage, especially for large datasets.
2. **Memory usage**: Tries can use a significant amount of memory, especially for large datasets.

3. **Insertion and deletion**: Insertion and deletion operations can be slow for large datasets.