Lab Manual (24 May 2023)

Topics Covered

1. Tries
    a. What are Tries
    b. Insertion in Tries
    c. Search in Tries
    d. Deletion in Tries
    e. Basic question related to Tries
2. External Sort
    a. Short Recap
    b. Try out: create a file filled with random data whose size is almost as large as main memory - show that time to read/write to disk dominates total runtime - actual in-memory sort doesn't take as much time
3. Radix Sort
    a. LSD to MSD (Implement Code)
    b. MSD to LSD (Implement Code)

▼ External Sort

1. Most of the internal sorting algorithms take advantage of the fact that memory (RAM) is directly addressable.
2. If the input is on a tape (Hard disk), then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially.
3. Even if the data are on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.
4. In most of the cases the time required to read and write to the tape is much longer than the time taken to sort the numbers in-memory

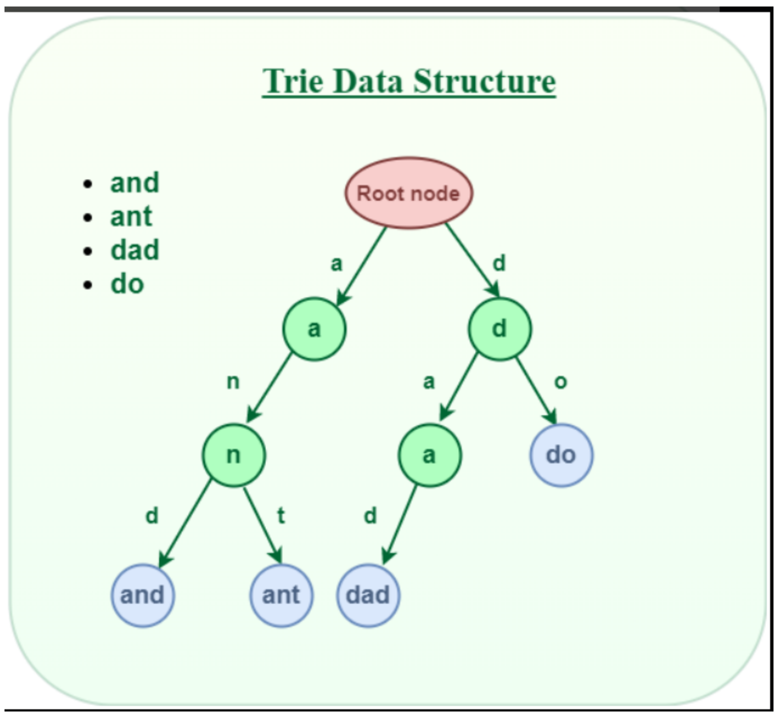
▼ Instructions to run the code

1. run `python3` `create.py` to create a file of 10^7 integers. The file should should be approximately $100\ MB$
2. run `g++ external.cpp` to sort the file using external sort. This should run for approximately $10\ secs$.
3. Observe and compare the time taken to read and write from the files $(8\ secs)$ and the time taken to sort in-memory elements .
4. Comment out `line 127` to observe the temporary files created during the external sort.

▼ Tries

1. A `Trie` is a specialized tree based data structure that is used for efficient pattern-matching and for efficient retrieval of strings. It is also known as digital tree (or) prefix-tree.
2. Some important applications are like **storing large amount of strings, spell-checking programs** and also **for tasks which involves dictionary like operations.**

Note: Unlike binary search trees (BST) or other common tree structures, where each node represents a single element, a `Trie` node represents a prefix or a complete string.



▼ Important Characteristics

1. **Structure of `Trie` Node :** Each node in a `Trie` consists of multiple pointers, often implemented using an array or a hash map, to its child nodes. The number of pointers typically corresponds to the size of the alphabet being used.

```
// Trie node
struct TrieNode
{
     struct TrieNode *children[ALPHABET_SIZE];
     // isEndOfWord is true if the node
     // represents end of a word
     bool isEndOfWord;
};
```

Note : A `Trie` node field **`isEndOfWord`** is used to distinguish the node as the end of the word node. It is false when we initialize the `TrieNode`.

2. **Prefixes and Complete Words:** A path from the root to a node represents a prefix or a complete word. The characters along the path form the string associated with that node.

3. **Leaf Nodes:** Leaf nodes represent complete words and often store additional information, such as associated values or frequencies.

▼ Basic Operations

The basic operations performed by tries are Insertion, Search and Deletion. We will see the implementation of these operations.

▼ Insertion

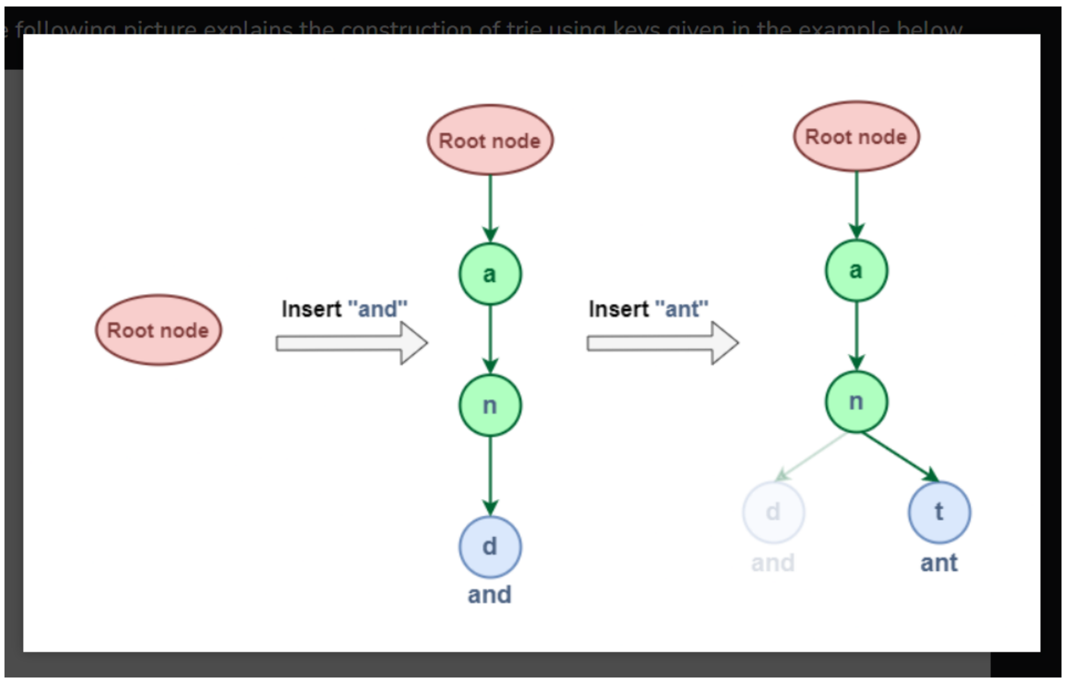
Algorithm:

1. Start at the root of the `Trie`.
2. For each character in the string, check if there is a child node corresponding to that character:
    - If a child node exists, move to that child node.
    - If a child node doesn't exist, create a new node and link it as a child of the current node at the corresponding character.
3. After iterating through all the characters in the string, mark the last node as the end of a word.

Pseudocode:

```
// Function to insert a string into the trie
insert(word):
    current = root
    for each character in word:
        index = character - 'a'
        if current.children[index] is null:
            current.children[index] = new Node
        current = current.children[index]
    current.isEndOfWord = true
```

The following picture explains the construction of `Trie` using keys given in the example below.

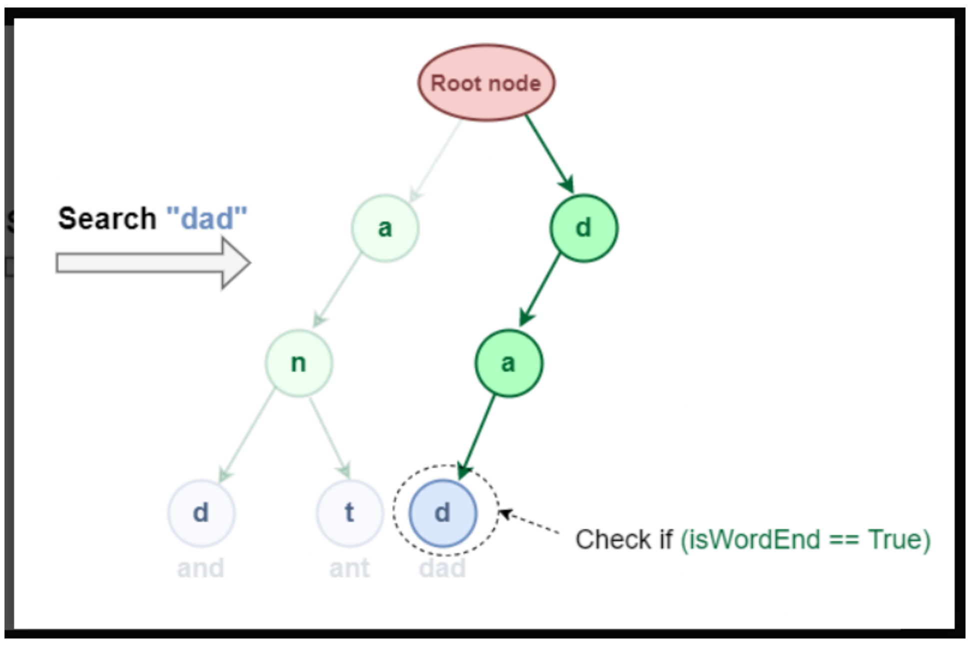


▼ Search

1. Searching for a key is similar to the insert operation. However, It only **compares the characters and moves down**.
2. The search can terminate due to the **end of a string** or **lack of key** in the `Trie`.
   - In the first case, if the `isEndofWord` field of the last node is true, then the key exists in the `Trie`.
   - In the second case, the search terminates without examining all the characters of the key, since the key is not present in the `Trie`.

Pseudocode:

```
// Function to search for a string in the trie
    bool search(word):
        current = root
        for each character in word:
            index = character - 'a'
            if current.children[index] is null:
                return false
            current = current.children[index]
        return current != null && current.isEndOfWord
```



▼ Delete

How to delete?

Step 1: Search for the key to be deleted:

- Start at the root of the `Trie`.
- Traverse the `Trie` character by character, following the path corresponding to the key to be deleted.
- If, at any point, a character is not found in the `Trie`, it means the key does not exist, and the delete operation terminates.

Step 2: Mark the node as non-leaf:

- Once the key is found, mark the last character node of the key as a non-leaf node. This is equivalent to changing `isEndofWord` to false. This indicates that the key is no longer present in the `Trie`.

Step 3: Delete key recursively:

- Starting from the last character node marked as non-leaf, recursively traverse back up in the `Trie`.
- For each node encountered, check if it has any children or if it is marked as a non-leaf node. If neither condition is true, delete the node.
- Continue this process until reaching the root of the `Trie` or until encountering a node with children or marked as a non-leaf node.

Pseudocode:

```
delete(key):
    current = root
    deleteHelper(current, key, 0)

deleteHelper(node, key, depth):
    if node is None:
        return node
    if depth == len(key):
        node.isLeaf = False
        if isNodeEmpty(node):
            node = None
        return node
    index = getCharIndex(key[depth])
    node.children[index] = deleteHelper(node.children[index], key, depth+1)
    if isNodeEmpty(node) and node.isLeaf == False:
        node = None
    return node

isNodeEmpty(node):
    for i in range(ALPHABET_SIZE):
        if node.children[i] is not None:
            return False
    return True

getCharIndex(char):
    return ord(char) - ord('a')
```

- The `delete` function initiates the delete operation by calling the `deleteHelper` function with the root node and the key to be deleted.
- The `deleteHelper` function is a recursive function that performs the delete operation.
- It checks if the current node is null and if the depth has reached the length of the key. If so, it marks the node as a non-leaf node and checks if the node can be deleted.
- If the depth has not reached the length of the key, it recursively calls the `deleteHelper` function on the appropriate child node based on the character index.
- After the recursive call, it checks if the node can be deleted.
- The `isNodeEmpty` function checks if a node is empty, i.e., it has no children.
- The `getCharIndex` function calculates the index of a character in the trie's children array.

Examples

1. **Example 1**

   Consider a trie containing the following keys: "apple," "apply," "banana," and "band." To delete the key "apple," the following steps are performed:

   a. Search for the key "apple." The characters 'a', 'p', 'p', 'l', 'e' are found in the trie.

   b. Mark the last character node 'e' as a non-leaf node.

   c. Delete key recursively by traversing back up the trie.

   d. The nodes 'e', 'l', 'p', 'p', and 'a' are deleted if they have no children.

2. Example 2:
   Consider a trie containing the following keys: "cat," "cats," "cut," and "cup." To delete the key "cup," the following steps are performed:

   a. Search for the key "cup." The characters 'c', 'u', 'p' are found in the trie.

   b. Mark the last character node 'p' as a non-leaf node.

   c. Delete key recursively by traversing back up the trie. The nodes 'p' and 'u' are deleted if they have no children.

▼ Practice

Question: Prefix Finder

Given a dictionary of strings and a string `target`, tell if `target` is a prefix of any word in the dictionary or not.

Consider the following example:

- Input: `["apple", "band", "bond"]` and given a target `"app"`
- Output: `"YES"`

Consider the following example:

- Input: `["apple", "band", "bond"]` and given a target `"bx"`
- Output: `"NO"`

Question: Autocomplete Feature

You are building an autocomplete feature for a text editor. The autocomplete feature should suggest completions for a given prefix based on a dictionary of words. You decide to implement this feature using a `Trie` data structure.

Implement a function `autocomplete(prefix, dictionary)` that takes a prefix string and a list of words in the dictionary and returns a list of all possible completions for the given prefix.

Example:
Consider the following dictionary of words: ["apple", "apply", "banana", "band", "cat", "cats"]

- Input: `autocomplete("ap", ["apple", "apply", "banana", "band", "cat", "cats"])`
- Output: `["apple", "apply"]`

The function should return all words in the dictionary that start with the prefix "ap."

Implement the `autocomplete` function using a `Trie` data structure and demonstrate its usage with multiple test cases.

Note: You can assume that the dictionary does not contain duplicate words.

Helper code to read `dictionary.txt`

```
#include <stdio.h>
#include <string.h>

int main()
{
    char buffer[20];
    FILE* fp = fopen("dictionary.txt", "r");

    while (fscanf(fp, "%s", buffer) == 1)
    {
        printf("%ld %s\n", strlen(buffer), buffer);
    }
```

```
    fclose(fp);
    return 0;
}
```

Question: Longest Common Prefix

Given a dictionary of strings, find the longest common prefix

Consider the following example:

- Input: `["applecherry", "appleby", "applecan"]`
- Output: `"apple"`

Question: Lexicographic Sorting of Strings

Given a set of strings, return them in lexicographic order using `Trie`.

1. Calculate time and space complexity.
2. Compare with lexicographic sorting using merge sort.