DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJ19ITL502 **DATE:4/10/22**
COURSE NAME: Advanced Data Structures Laboratory **CLASS/Div: A**
SAP ID:60003200076 **Academic Year 2022-23**

EXPERIMENT NO. 2

CO/LO: LO1

AIM: To Implement standard Tries

Theory:

The word "Trie" is an excerpt from the word "retrieval".

Trie is a sorted tree-based data-structure that stores the set of strings. It has the number of pointers equal to the number of characters of the alphabet in each node. It can search a word in the dictionary with the help of the word's prefix.

For example, if we assume that all strings are formed from the letters 'a' to 'z' in the English alphabet, each trie node can have a maximum of 26 points.Trie is also known as the digital tree or prefix tree. The position of a node in the Trie determines the key with which that node is connected.

Properties of the Trie for a set of the string:

- The root node of the trie always represents the null node.
- Each child of nodes is sorted alphabetically.
- Each node can have a maximum of 26 children (A to Z).
- Each node (except the root) can store one letter of the alphabet.

Technology stack used: Python

Algorithm:

Algorithm insert(root,word):

Iterate over each character in the word:

If char already exists:

Check next char

Check if last char

Else:

Create new node with value of char and add it to the char position child

Check if last char

Check next char

Algorithm search(root,word):

Iterate over each character in the word:

If char is present in the children array of root:

Check next char

Else:

Return “word doesn’t exist”

If the pointer ends up at node with isend==True:

Return “word exist”

Else:

Return “word doesn’t exist”

Algorithm deletet(root,word):

Search(word)

If word exists:

Change the isend of last node char to False

Return “word deleted”

Else:

Return “word doesn’t exist”

Program:

```
class node:
    def __init__(self,char):
        self.char=char
        self.end=0
        self.pointer=[]
        for i in range(0,26):
            self.pointer.append(-1)

root=node(0)
#print(ord("b")-ord("a"))
def printnode(root):
    current=root
    n=0
    while current.end!=1:
        for i in current.pointer:
            if(i!=-1):
                print(i.char)
                current=current.pointer[current.pointer.index(i)]


def insert(root,data):
    current=root
    k=0
    for i in data:
        idx=ord(i)-ord("a")
        if(current.pointer[idx]==-1):
            n=node(i)
            if(k==len(data)-1):
                n.end=1
            current.pointer[idx]=n
            #print(current.pointer)
            current=current.pointer[idx]
        else:
            current=current.pointer[idx]
            if(k==len(data)-1):
                current.end=1

        k=k+1

def searchnode(root,data):
    current=root
    for i in data:
        idx=ord(i)-ord("a")
        if(current.pointer[idx]==-1):
```

```
            print(data," doesent exist in trie")
            return
        elif(current.pointer[idx].char==i):
            #print("found ",i)
            current=current.pointer[idx]

    if(current.end==1):
        print(data," exists in trie")
    else:
        print(data," doesent exist in trie")

def delnode(root,data):
    current=root
    for i in data:
        idx=ord(i)-ord("a")
        if(current.pointer[idx]==-1):
            print(data," doesent exist in trie cant delete")
            return
        elif(current.pointer[idx].char==i):
            #print("found ",i)
            current=current.pointer[idx]

    if(current.end==1):
        current.end=0
        print("node deleted")
    else:
        print(data," doesent exist in trie cant delete")



loop=True
choice=1
print("\n1==insert\t2==search\t3==delete\t4==exit\n")
while(loop):
    if choice==1:
        ele=input("enter the node to insert : ")
        insert(root,ele)
        choice=-1
    elif choice==2:
        ele=input("enter the node to search : ")
        searchnode(root,ele)
        choice=-1
    elif choice==3:
        ele=input("enter the node to delete : ")
        delnode(root,ele)
        choice=-1
```

```
    elif choice==-1:
        print("\n1==insert\t2==search\t3==delete\t4==exit")
        choice=int(input("enter your choice : "))
    elif choice==4:
        loop=False
```

Analysis:

For a word of length n

Insertion:

O(n) : In each iteration of the algorithm, we either examine or create a node in the trie till we reach the end of the key. This takes only n operations.

Searching:

O(n) : In each iteration of the algorithm, we examine if the current char of the word exists as a child of the previous char

Deletion:

O(n) : we first search the word in trie which takes O(n) then change the isend to False

Output:

```
PS C:\Users\SHREE RAM\Desktop\ads> python -u "c:\Users\SHREE RAM\Desktop\ads\tries.py"

1==insert       2==search       3==delete       4==exit

enter the node to insert : amma

1==insert       2==search       3==delete       4==exit
enter your choice : 1
enter the node to insert : am

1==insert       2==search       3==delete       4==exit
enter your choice : 1
enter the node to insert : hello

1==insert       2==search       3==delete       4==exit
enter your choice : 2
enter the node to search : am
am  exists in trie

1==insert       2==search       3==delete       4==exit
enter your choice : 2
enter the node to search : amma
amma  exists in trie

1==insert       2==search       3==delete       4==exit
enter your choice : 3
enter the node to delete : am
node deleted

1==insert       2==search       3==delete       4==exit
enter your choice : 2
enter the node to search : am
am  doesent exist in trie

1==insert       2==search       3==delete       4==exit
enter your choice : 4
PS C:\Users\SHREE RAM\Desktop\ads>
```

Conclusion:

1. With Trie, we can insert and find strings in *O(L)* time where *L* represent the length of a single word. This is obviously faster than BST. This is also faster than Hashing because of the ways it is implemented. We do not need to compute any hash function. No collision handling is required (like we do in open addressing and separate chaining)

2. Another advantage of Trie is, we can easily print all words in alphabetical order which is not easily possible with hashing.

3. We can efficiently do a prefix search (or auto-complete) with Trie

Thus we have implemented standard trie