Struktur Data *Trie* dalam Aplikasi *Sotfware Keyboard* yang Dipersonalisasi

Johanes Lee - 13521148[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13521148@std.stei.itb.ac.id

Abstract*—Tree data structure has been widely used in many applications. There are various modifications on the data structure that is made to adapt certain problems. One modification results in a data structure called retrieval tree or trie. This data structure provides some advantages in string processing, including having approximately constant complexity in its searching algorithm. That is, the complexity of string searching in trie is not affected by the number of strings stored in it. With this benefit in mind, trie can be used in implementing auto-complete and word recommendation algorithm, especially in software keyboard.

***Keywords*—Auto Complete, Searching, Software Keyboard, Trie.**

I. PENDAHULUAN

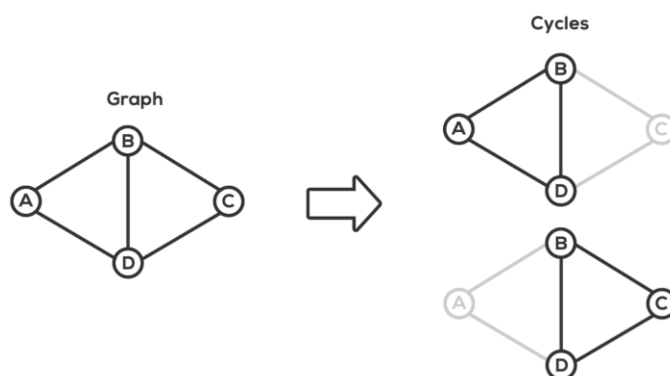
Struktur data *tree* sudah banyak digunakan dalam berbagai aplikasi. Struktur data ini salah satunya memiliki keunggulan dalam operasi *searching*, seperti pada struktur data *binary search tree*. Namun, aplikasi *tree* dalam operasi *searching* tidak hanya terbatas pada tipe data numerik. Struktur data *retrieval tree* atau *trie*, yang menggabungkan *tree* dengan *hash table* pada salah satu implementasinya, merupakan salah satu struktur data paling sederhana dan ideal dalam penerapan fitur pencarian berbasis teks atau *string* [1]. Struktur data *trie* memungkinkan penerapan proses pencarian string dengan kecepatan yang tidak dipengaruhi oleh besar data pada *trie* tersebut (tetapi dipengaruhi oleh panjang string yang dicari). Dengan keuntungan tersebut, struktur data ini dapat digunakan dalam pencarian dan fitur *auto complete* pada *seach engine* ataupun aplikasi berbasis teks lainnya.

Keuntungan-keuntungan yang disediakan *trie* juga memungkinkan struktur data tersebut digunakan dalam pembuatan *software keyboard* yang bersifat prediktif (melalui fitur *auto complete* dan sistem rekomendasi). Penerapan ini ditujukan untuk mengurangi usaha yang diperlukan dalam mengetik suatu kata dan meningkatkan efisiensi waktu pengetikan. Modifikasi pada *trie* (salah satunya berupa penambahan informasi pada setiap *node* ataupun hanya *node* tertentu) juga memungkinan pengembangan *software keyboard* yang dipersonalisasi, yaitu bergantung pada kata-kata yang sering digunakan pengguna dan tidak bergantung pada suatu kamus statis yang didefinisikan. Penerapan ini juga dapat membuat suatu *software keyboard* bersifat adaptif dengan bahasa yang digunakan pengguna, berbeda dengan penggunaan kamus statis yang umumnya berpaku pada satu bahasa [2].

II. LANDASAN TEORI

A. Graf

Graf (*graph*) adalah struktur data berbasis *node* dengan suatu hubungan antar-*node* tersebut [1]. *Node* di dalam graf sering disebut simpul (*vertex*) sedangkan garis yang menghubungkan dua simpul disebut sisi (*edge*). Selain itu, dikenal juga lintasan, yaitu barisan yang saling bergantian antara simpul dan sisi, berawal dari simpul awal v_0 menuju simpul tujuan v_n, sedemikian sehingga (untuk edge yang dilambangkan dengan e) $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, ..., $e_n = (v_{n-1}, v_n)$ merupakan sisi-sisi graf tersebut [3]. Siklus (*cycle*) atau sirkuit (*circuit*) adalah lintasan yang berawal dan berakhir pada simpul yang sama (ilustrasi ditunjukkan pada gambar 1). Suatu graf dikatakan terhubung apabila graf tersebut memiliki lintasan dari v_i menuju v_j untuk setiap pasangan simpul pada graf tersebut [3].

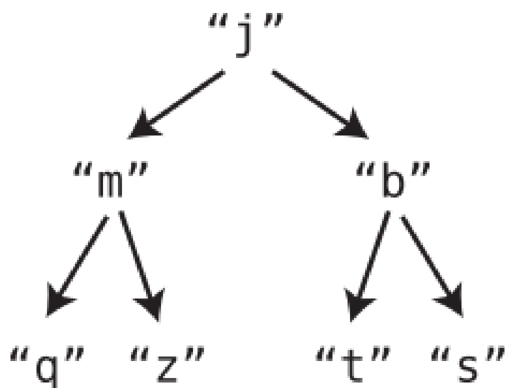


Gambar 1. Sirkuit dalam graf
(Sumber: https://javascript.plainenglish.io/finding-simple-cycles-in-an-undirected-graph-a-javascript-approach-1fa84d2f3218, diakses pada 2 Desember 2022)

B. Pohon

Pohon (*tree*) adalah sebuah graf yang terhubung dan tidak mengandung sirkuit [4]. Di dalam struktur data ini, dikenal *rooted tree*, yaitu pohon dengan salah satu simpul merupakan akar dan sisi-sisinya diberi arah sehingga menjadi graf berarah [5]. Akar umumnya digambarkan sebagai simpul teratas. Pada gambar 2, simpul j disebut akar, simpul m dan b adalah anak (*child*) dari simpul j, serta simpul j juga merupakan orang tua kedua simpul tersebut (hubungan orang tua dan anak ini juga terdapat pada simpul m dengan dua simpul q dan z serta simpul

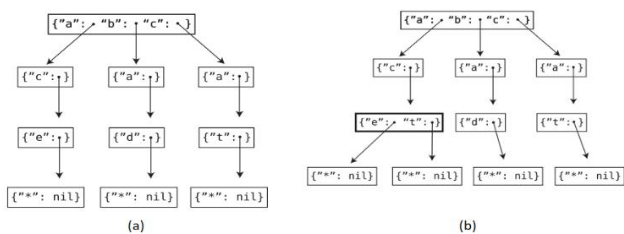
b dengan dua simpul t dan s). Selain itu, simpul q, z, t, maupun s disebut simpul daun (*leaf node*), yaitu simpul yang tidak memiliki anak. (jumlah anak pada suatu simpul di dalam *rooted tree* juga disebut derajat simpul sehingga simpul daun memiliki derajat nol [5]).



Gambar 2. Representasi struktur data *Tree*
(Sumber: Wengrow, J., *A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.)*, Pragmatic Bookshelf, 2020, pp. 248)

C. Trie

Retrieval tree atau *trie* adalah jenis pohon yang ideal untuk algoritma fitur berbasis teks atau *string* [1]. Dalam salah satu implementasinya yang berbasis *node*, masing-masing *node* pada *trie* sederhana dapat menyimpan suatu *hash table* dengan kunci (*key*) berupa suatu karakter (umumnya dibatasi pada karakter alfabet) serta *value* berupa alamat *node* selanjutnya yang berhubungan dengan karakter tersebut (*hash table* dapat digantikan dengan struktur data lain tetapi pemilihan *hash table* memungkinkan pengaksesan alamat *node* selanjutnya berdasarkan suatu karakter dengan hanya satu aksi).



Gambar 3. (a) Representasi struktur data *trie* dan (b) Penyimpanan kata baru pada *trie*
(Sumber: Wengrow, J., A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.), Pragmatic Bookshelf, 2020, pp. 308)

Struktur data *trie* berfokus pada prefiks satu atau lebih string yang sama. Gambar 3*a* mengilustrasikan struktur data *trie* dengan implementasi berbasis *node* berisi *hash table* sedangkan gambar 3*b* menggambarkan struktur data tersebut setelah penambahan *string* "act". Pada proses penyimpanan *string* baru tersebut, hanya ditambahkan pasangan *key-value* baru untuk karakter *t* pada *string* "act" karena prefiksnya, yaitu "ac", sudah disimpan di dalam *trie* tersebut. Selain itu, setiap akhir kata yang disimpan ditandai dengan karakter "*". Hal ini diperlukan untuk mengidentifikasi dua atau lebih kata dengan prefiks yang sama [1].

III. Aplikasi *Trie* dalam Fitur *Auto Complete* dan Rekomendasi untuk *Software Keyboard* yang Dipersonalisasi

A. Implementasi Struktur Data

Struktur data berbasis *node* umumnya menggunakan *pointer* dalam implementasinya. Namun, bahasa pemrograman tidak selalu menyediakan penggunaan *pointer* (seperti Python). Selain *pointer*, *class object* (umum digunakan pada *Object Oriented Programming* atau OOP) juga dapat digunakan dalam merepresentasikan *node* pada *tree* ataupun struktur data berbasis *node* lainnya.

Hash table (ataupun *dictionary* dalam bahasa pemrograman Python) memungkinkan proses pengaksesan yang sangat cepat terhadap suatu nilai (*value*) dengan *key* tertentu, yaitu dengan kompleksitas $O(1)$. Struktur ini dapat dimanfaatkan dalam penerapan struktur data *trie* untuk mendukung efisiensi proses pencarian. Dengan demikian, setiap *node* pada *trie* dapat berisi sebuah *hash table* dengan kunci berupa karakter-karakter alfabet dan *value* berupa *instance* sebuah *class* yang merepresentasikan anak (*child*) dari *node* tersebut. *Trie class* juga dapat dibuat untuk menyimpan *root node* sebagai *attribute* serta mendefinisikan metode-metode yang dibutuhkan untuk struktur data ini [1]. Kode Python di bawah menunjukkan implementasi *trie node* dan *trie class* yang dapat digunakan dalam aplikasi fitur *auto complete* dan sistem rekomendasi sederhana. Perlu diperhatikan bahwa anak setiap *node* pada struktur data *trie* tidak dibatasi jumlahnya, berbeda dengan struktur data pada *binary search tree*.

```
class TrieNode:
  def __init__(self):
    self.children = {}

class Trie:
  def __init__(self):
    self.root = TrieNode()
```

(Sumber: Wengrow, J., A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.), Pragmatic Bookshelf, 2020, pp. 307)

B. Pencarian String dalam Trie

Pencarian dalam struktur data *trie* dapat menentukan apakah suatu *string* (berupa suatu kata) yang dicari merupakan suatu kata utuh, prefiks sebuah kata utuh, ataupun kata baru (tidak ditemukan dalam pencarian tersebut). Dalam proses pencarian, perlu dilakukan iterasi terhadap masing-masing karakter dalam *string* yang sedang diproses, mulai dari karakter pertama *string* tersebut. Perlu diperhatikan bahwa suatu *string* juga dapat berupa kata utuh sekaligus menjadi prefiks suatu kata utuh lainnya yang lebih panjang.

Dalam algoritma pencarian pada *trie,* program memeriksa

(mulai dari *root*) apakah ada *child node* yang memiliki kunci yang sama dengan karakter yang sedang diproses. Program kemudian akan memproses karakter selanjutnya dan memeriksa *child node* tersebut jika ada sedangkan proses pencarian berhenti jika tidak ditemukan *child node* tersebut sehingga *string* yang sedang diproses dikategorikan sebagai kata baru. Program akan mengulang proses tersebut untuk setiap karakter yang belum diproses dan berhenti ketika semua karakter sudah diproses. Apabila semua karakter selesai diproses dan *node* saat ini tidak memiliki simbol penanda akhir kata sebagai salah satu kunci, *string* yang sedang diproses akan dikategorikan sebagai prefiks satu atau lebih kata utuh yang ada pada *trie*. Namun, jika ditemukan simbol tersebut pada kondisi yang sama, *string* tersebut ditategorikan sebagai kata utuh yang sudah tersimpan di dalam *trie*. Berikut merupakan implementasi algoritma pencarian untuk struktur data *trie* dalam bahasa Python.

```
def search(self, word):
  currentNode = self.root
  for char in word:
    if currentNode.children.get(char):
      currentNode = (
          currentNode.children[char] )
    else:
      return None

  return currentNode
```

(Sumber: Wengrow, J., A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.), Pragmatic Bookshelf, 2020, pp. 313)

Kompleksitas algoritma pencarian untuk struktur data *trie* yang telah dijelaskan adalah $O(K)$ dengan K merepresentasikan panjang *string* yang sedang diproses [1]. Dengan demikian, kecepatan pencarian *string* tidak dipengaruhi oleh banyak data yang ada pada *trie*. Hal ini menyebabkan kecepatan algoritma tidak akan menurun seiring pertambahan data akibat penambahan kata-kata baru. Bahkan, kompleksitas yang diperoleh dapat tergolong lebih cepat dibandingkan kompleksitas $O(\log N)$ dengan N merupakan jumlah data (kata yang disimpan) ketika menggunakan *binary search* pada *ordered array*.

C. Penambahan String dalam Trie

Proses penambahan (*insertion*) pada *trie* diawali dengan proses yang serupa dengan proses pencarian. Perbedaan proses terletak pada saat tidak ditemukan *child node* berdasarkan karakter yang sedang diproses. Pada kondisi tersebut, dibuat kunci dan *node* (atau *class instance*) baru sebagai *value*, berpindah ke *node* baru tersebut, kemudian memproses karakter selanjutnya serta mengulangi kembali proses tersebut hingga semua karakter selesai diproses. Pada akhir pemrosesan, perlu ditambah simbol penanda akhir kata sebagai kunci pada *node* yang paling terakhir dikujungi. Kompleksitas yang diperoleh sama dengan proses pencarian, yaitu $O(K)$ [1]. Kode Python berikut menggambarkan proses penambahan *string* di dalam struktur data *trie*.

```
def insert(self, word):
  currentNode = self.root
  for char in word:

    if currentNode.children.get(char):
      currentNode = (
          currentNode.children[char] )
    else:
      newNode = TrieNode()
      currentNode.children[char] = (newNode)
      currentNode = newNode

  # char “*” is used as mark
  currentNode.children["*"] = None
```

(Sumber: Wengrow, J., A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.), Pragmatic Bookshelf, 2020, pp. 319)

D. Fitur Auto Complete dan Rekomendasi Sederhana

Fitur *auto complete* dan rekomendasi sederhana dapat dibuat dengan mencari kata-kata utuh berdasarkan suatu *string* prefiks tertentu. Proses pencarian ini dapat menggunakan algoritma *Depth-First Search* (DFS) dengan prinsip rekursif. Proses pencarian dimulai dari *node* tempat algoritma pencarian *string* pada *trie* berhenti saat pencarian string *prefiks* tersebut. Berikut merupakan implementasi algoritma pencarian kata-kata rekomendasi pada *trie* dalam bahasa Python.

```
def collectAllWords(self, node=None, word="",
words=[]):
  currentNode = node or self.root

  for key, childNode in (
        currentNode.children.items()):

    if key == "*":
      words.append(word)
    else:
      self.collectAllWords(childNode,
          word + key, words)

  return words
```

(Sumber: Wengrow, J., A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.), Pragmatic Bookshelf, 2020, pp. 321)

E. Peningkatan Kinerja Fitur Auto Complete dan Sistem Rekomendasi Sederhana

Fitur *auto complete* dan rekomendasi sederhana ditujukan untuk membantu kecepatan pengetikan suatu kata atau kalimat. Dengan demikian, fitur yang dibuat tidak boleh menyulitkan pengguna dengan kerumitannya. Salah satu yang dapat dilakukan untuk meningkatkan kemudahan penggunaan adalah dengan membatasi kata-kata rekomendasi yang ditampilkan ke pengguna.

Rekomendasi dapat ditampilkan berdasarkan popularitas kata

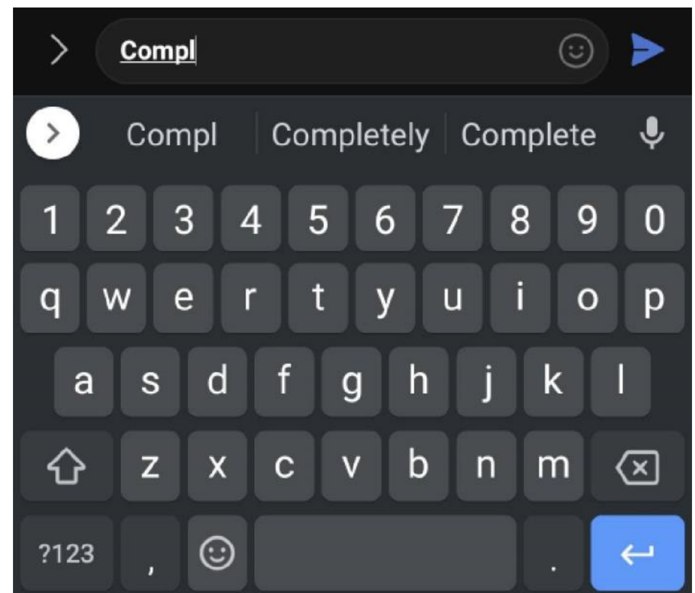
yang tersimpan di dalam *trie* [1]. Dalam implementasinya, diperlukan penyimpanan data nilai popularitas tersebut di dalam *trie*. *Node* yang menyimpan penanda akhir kata dapat digunakan untuk keperluan tersebut. Hal ini disebabkan pada implementasi sebelumnya nilai yang ditunjuk oleh kunci "*" tidak menyimpan data yang berarti. Data ini dapat diganti dengan nilai popularitas kata yang berakhir pada *node* tersebut. Dengan demikian, sistem rekomendasi yang dibuat dapat mengakses nilai popularitas masing-masing kata yang direkomendasikan dan kemudian hanya menampilkan beberapa kata dengan nilai popularitas tertinggi. Berikut modifikasi implementasi fungsi collectAllWords yang dapat dilakukan dalam realisasi ide tersebut. Selain itu, pada *insertion*, *node* terminal dapat menyimpan *value* bernilai 1 sebagai nilai popularitas awal kata yang baru disimpan.

```
def collectAllWords(self, node=None, word="",
words=[]):
  currentNode = node or self.root

  for key, childNode in (
       currentNode.children.items()):

    if key == "*":
      # Store word with its popularity value
      words.append(
           (currentNode.children["*"], word))

    else:
      self.collectAllWords(childNode,
         word + key, words)

  # sort recommendations based on popularity
  words.sort(reverse=True)

  return words
```

F. Fitur Auto Complete dan Sistem Rekomendasi pada Software Keyboard

Aplikasi *software keyboard* yang dipersonalisasi dapat dibuat dengan penerapan struktur data *trie*. Salah satu ide yang dapat direalisasikan adalah dengan menggunakan *trie* yang pada mulanya kosong (tidak ada kata yang disimpan) untuk masing-masing pengguna. *Trie* tersebut kemudian akan terisi selama penggunaan aplikasi dengan cara menyimpan kata-kata yang diketik pengguna. Dengan pendekatan ini, *trie* yang terbentuk akan bergantung pada kata-kata yang sering digunakan pengguna dan dengan demikian aplikasi akan bersifat adaptif dengan bahasa pengguna. Popularitas suatu kata dapat ditambahkan setiap pengguna selesai mengetik kata yang bersangkutan.

Sistem rekomendasi dapat memanfaatkan *User Interface* (UI) aplikasi dalam menampilkan kata-kata populer yang ditemukan. Tombol rekomendasi umum digunakan dalam menampilkan rekomendasi-rekomendasi tersebut. Tombol tersebut juga dapat digunakan sebagai salah satu cara implementasi fitur *auto complete*. Pengguna dapat menekan kata yang direkomendasikan sehingga kata yang belum selesai diketik akan langsung diganti dengan kata utuh yang dipilih. Implementasi lainnya dapat dilakukan dengan langsung mengganti kata yang sedang diketik dengan kata terpopuler yang ditemukan ataupun yang diperoleh dengan algoritma tambahan (seperti dengan cara menghitung probabilitas kata yang ingin diketik pengguna [2]) sehingga fitur *auto complete* bekerja tanpa aksi tambahan yang harus dilakukan pengguna (seperti menekan tombol). Kedua contoh implementasi fitur *auto complete* ini dapat terlihat pada *software keyboard* yang ada pada hampir semua *smartphone*.



Gambar 4. Sistem rekomendasi pada *software keyboard* di dalam *smartphone*

Ide fitur *auto complete* dan sistem rekomendasi sederhana yang telah dibahas diimplementasikan dalam program bahasa Python di bawah. Program ini menyimulasikan (secara kasar) proses pengetikan di dalam *software keyboard*. Di dalam program yang telah dibuat, pengguna dapat mengetik satu atau lebih karakter dalam satu kali perulangan (*loop*). Semua karakter yang telah di ketik (sejak awal perulangan) dicatat oleh program dan karakter-karakter tambahan yang diketik pengguna merupakan karakter-karakter lanjutan dari seluruh karakter yang tercatat. Pengguna dianggap memfinalisasi seluruh karakter yang telah diketik menjadi kata ketika memasukkan input kosong (dilakukan dengan memasukkan *enter*).

Program menampilkan maksimal 3 kata rekomendasi dengan popularitas tertinggi dengan prefiks masing-masing kata tersebut merupakan karakter-karakter yang telah diketik pengguna (sebelum finalisasi kata). Pengguna dapat memasukkan *tab* untuk melakukan *auto complete*, yaitu mengganti seluruh karakter yang telah diketik menjadi kata utuh yang direkomendasi dengan popularitas tertinggi serta memfinalisasinya. Jika tidak ada rekomendasi yang ditampilkan, karakter-karakter yang telah diinput bukan merupakan prefiks karakter apapun yang tersimpan di dalam *trie* dan masukan *tab* akan berfungsi sama dengan *enter*, yaitu memfinalisasi kata berupa seluruh karakter yang telah diketik tanpa *auto complete*.

```
# Implementasi Fitur Auto-Complete dan Sistem Rekomendasi Sederhana

class TrieNode:
  def __init__(self):
    self.children = {}

class Trie:
  def __init__(self):
    self.root = TrieNode()

  def insert(self, word):
    currentNode = self.root
    for char in word:

      if currentNode.children.get(char):
        currentNode = (currentNode.children[char])
      else:
        newNode = TrieNode()
        currentNode.children[char] = (newNode)
        currentNode = newNode

    # char "*" is used as mark
    currentNode.children["*"] = 1

  def search(self, word):
    currentNode = self.root
    for char in word:
      if currentNode.children.get(char):
        currentNode = (currentNode.children[char])
      else:
        return None

    return currentNode


  def collectAllWords(self, node=None, word="", words=[]):
    currentNode = node or self.root

    for key, childNode in (currentNode.children.items()):

      if key == "*":
        # Store word with its popularity value
        words.append((currentNode.children["*"] ,word))

      else:
        self.collectAllWords(childNode,
          word + key, words)

    # sort recommendations based on popularity (descending)
    words.sort(reverse=True)
    return words

def displayRecommendation(recommendations, count = 3):

  print("Recommendations: ", end="")
  iteration = min(count, len(recommendations))
  for i in range(iteration):

    print(recommendations[i], end="")
    if (i < iteration - 1):
      print(" | ", end="")

    else:
      print()

def typeKeyboard():
```

```
    newTrie = Trie()
    currentSubTrie = Trie()

    newTrie.insert("ace")
    newTrie.insert("bad")
    newTrie.insert("cat")
    newTrie.insert("act")
    newTrie.insert("complete")
    newTrie.insert("completely")

    while(True):

      currentSubTrie.root = newTrie.root
      currentTypedWord = ""
      chars = ""
      unprocessedString = ""
      autoCompleteWord = ""

      print("\nYour Input:", currentTypedWord, end="")
      chars = input()
      notFound = False

      while(not (len(chars) == 0) and not(chars == "\t")):
        unprocessedString += chars

        if (not(notFound)):
          currentNode = currentSubTrie.search(chars)
          currentSubTrie.root = currentNode

        if (not(currentNode == None)):
          word = currentTypedWord + unprocessedString
          recommendations = currentSubTrie.collectAllWords(word=word, words=[])

          currentTypedWord += unprocessedString
          unprocessedString = ""
          displayRecommendation(recommendations)
          autoCompleteWord = recommendations[0][1]

        else:
          notFound = True
          print("[Word is not recognized in current trie]")
          currentTypedWord += chars
          autoCompleteWord = ""

        print("\nYour Input:", currentTypedWord, end="")
        chars = input()

      if (chars == "\t" and len(autoCompleteWord) > 0):
        print("Your word is auto-completed as", autoCompleteWord)
        currentNode = currentSubTrie.search(autoCompleteWord.removeprefix(currentTypedWord))
        currentNode.children["*"] += 1

      elif (len(currentTypedWord) > 0):
        if(len(unprocessedString) > 0 or not(currentSubTrie.root.children.get("*"))):
          newTrie.insert(currentTypedWord)
          print(currentTypedWord, "is added to trie!\n")

        else:
          currentNode.children["*"] += 1
          print("You typed existing word in trie!")

typeKeyboard()
```

Finalisasi suatu kata baru akan menyebabkan program menyimpan kata baru tersebut ke dalam *trie* dengan nilai popularitas awal bernilai 1. Selain itu, finalisasi kata yang sudah ada menyebabkan nilai popularitas kata tersebut bertambah. Dengan demikian, data pada *trie* akan terus diperbarui setiap pengguna melakukan finalisasi kata.

Di dalam fungsi typeKeyboard terlihat program terlebih dahulu memasukkan 6 kata berbeda, dengan penambahan 4 kata pertama ("ace", "bad", "cat", dan "act") bertujuan untuk mensimulasikan struktur data yang ditampilkan gambar 3*b*. Berikut tampilan *console* ketika program dijalankan dan dimasukkan karakter "a" diikuti masukan karakter "ct" setelahnya.

```
# input "a"
Your Input: a
Recommendations: (1, 'act') | (1, 'ace')

# input "ct"
Your Input: act
Recommendations: (1, 'act')

Your Input: act_
```

Rekomendasi ditampilkan dalam bentuk *tuple* berisi nilai popularitas beserta kata yang bersangkutan. Ketika pengguna memasukkan *enter* (tanpa karakter tambahan setelah masukan bernilai "act"), kata tersebut akan difinalisasi. Karena kata "act" sudah ada pada *trie*, efek finalisasi kata tersebut adalah menambah nilai popularitas kata yang bersangkutan (terlihat nilai popularitas kata yang difinalisasi bertambah menjadi 2 pada tampilan di bawah).

```
# input enter
Your Input: act
You typed existing word in trie!

# pengecekan rekomendasi setelah finalisasi
Your Input: a
Recommendations: (2, 'act') | (1, 'ace')

Your Input: a_
```

Tampilan di bawah menunjukkan kasus ketika pengguna memasukkan *tab* dengan tujuan melakukan *auto complete* terhadap seluruh karakter yang belum difinalisasi. Kata yang dipilih pada fitur *auto complete* merupakan kata dengan popularitas tertinggi (pada contoh tampilan di bawah, kedua kata memiliki nilai popularitas yang sama sehingga dipilih kata terdepan yang ditampilkan). Nilai popularitas kata utuh yang difinalisasi akan bertambah setelahnya. Selain itu, jika *string* yang sudah dimasukkan bukan merupakan *prefiks* kata apapun yang ada di dalam *trie*, finalisasi kata akan menyebabkan kata tersebut ditambahkan ke dalam *trie*.

```
# input "comp"
Your Input: comp
Recommendations: (1, 'completely') | (1,
'complete')

# input tab (auto complete)
Your Input: comp
Your word is auto-completed as completely

# pengecekan rekomendasi setelahnya
Your Input: co
Recommendations: (2, 'completely') | (1,
'complete')

# input "mpleted"
Your Input: completed
[Word is not recognized in current trie]

# input enter (finalisasi)
Your Input: completed
completed is added to trie!

# pengecekan rekomendasi
Your Input: co
Recommendations: (2, 'completely') | (1,
'completed') | (1, 'complete')

# input tab (auto complete)
Your Input: co
Your word is auto-completed as completely

# pengecekan rekomendasi
Your Input: com
Recommendations: (3, 'completely') | (1,
'completed') | (1, 'complete')

Your Input: com_
```

IV. Pengembangan Aplikasi

Proses pencarian kata yang akan direkomendasikan pada program yang telah dibuat memiliki waktu eksekusi yang bergantung pada jumlah simpul anak yang disimpan pada *trie* setelah *node* awal pencarian. Hal ini dapat menyebabkan adanya jeda waktu apabila kata yang disimpan di dalam *trie* berjumlah banyak padahal kompleksitas konstan (atau mendekati konstan) sangat dibutuhkan dalam aplikasi ini untuk mendukung kecepatan pengetikan pengguna. Salah satu solusi permasalahan ini adalah dengan menyimpan nilai popularitas pada setiap *node* (termasuk *node* bukan terminal). Nilai popularitas dapat diperbarui setiap *node* yang bersangkutan ditelusuri di dalam algoritma yang mencatat pengetikan pengguna. Dengan demikian, program dapat menggunakan nilai-nilai popularitas tersebut untuk menentukan jalur penulusuran yang dapat membawanya menuju kata dengan popularitas tertinggi. Hal ini membuat program tidak perlu mencari seluruh kemungkinan kata rekomendasi yang ada sehingga meningkatkan efisiensi program (tetapi juga meningkatkan penggunaan memori).

Program yang telah dibuat hanya dapat memprediksi kata yang memiliki prefiks berupa *string* yang diinput. Algoritma yang lebih kompleks akan dapat membuat program melakukan pemrosesan yang lebih luas, seperti pencarian kata homofon [6]. Pencarian ini dapat meningkatkan fleksibilitas aplikasi, terutama pada kasus ketika terjadi kesalahan ketik oleh pengguna yang menyebabkan *string* masukan tidak sesuai dengan prefiks kata yang dituju.

V. SIMPULAN

Struktur data *trie* memungkinkan proses pencarian kata yang sangat cepat dan tidak bergantung pada banyak kata yang disimpan. Keuntungan yang disediakan struktur data ini dapat dimanfaatkan dalam pembuatan fitur *auto complete* dan sistem rekomendasi dalam *software keyboard*. Penyimpanan data tambahan berupa nilai popularitas untuk setiap kata yang disimpan juga dapat meningkatkan kualitas sistem rekomendasi pada aplikasi tersebut. Namun, struktur data ini dapat membutuhkan banyak alokasi memori.

VI. UCAPAN TERIMA KASIH

Penulis menyampaikan puji syukur kepada Tuhan Yang Maha Esa atas berkat-Nya yang memungkinkan penulis menyelesaikan makalah ini. Penulis juga berterima kasih kepada dosen pengampu mata kuliah Matematika Diskrit Semester Ganjil 2022/2023 kelas 02, Ibu Fariska Zakhralativa Ruskanda, S.T., M.T., yang telah menyalurkan ilmu-ilmu yang digunakan dalam penulisan makalah ini. Tugas makalah ini telah menambah wawasan Matematika Diskrit yang lebih mendalam bagi penulis dan juga menambah pengalaman penulis dalam pembuatan makalah ilmiah.

REFERENCES

[1] Wengrow, J., *A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills (2nd ed.)*, Pragmatic Bookshelf, 2020, pp. 247–328.
[2] Sakkos, Panos & Kotsakos, Dimitrios & Katakis, Ioannis & Gunopulos, Dimitrios, Anima: Adaptive Personalized Software Keyboard, 2015.
[3] R. Munir, "Graf (Bag. 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf, diakses pada 2 Desember 2022.
[4] R. Munir, "Pohon (Bag. 1)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Pohon-2020-Bag1.pdf, diakses pada 2 Desember 2022.
[5] R. Munir, "Pohon (Bag. 2)", https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/Pohon-2021-Bag2.pdf, diakses pada 2 Desember 2022.
[6] Parmar, Vimal P. and Dr. C. K. Kumbharana, "Implementation of Trie Structure for Storing and Searching of English Spelled Homophone Words.", 2017.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2022