Trie

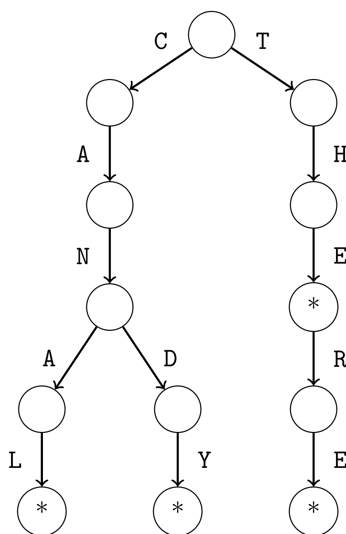
Jelena Hadži-Purić

februar 2019

1 Struktura podataka TRIE

Preksno drvo takodje poznato pod engleskim nazivom **trie** (od engleske reči reTRIEeval) je korensko stablo koje čuva skup niski (stringova). Svaka niska u skupu se skladišti kao lanac svojih karaktera, tako da olanačavanje počinje iz korena. Ako dve niske imaju zajednički prefiks, one takodje imaju i zajednički lanac u drvetu **trie**.

Na primer, uočimo sledeći trie:



Ovaj trie odgovara skupu {CANAL, CANDY, THE, THERE}. Znak * unutar čvora kazuje da se niska u skupu završava u tom čvoru. Takav (terminirajući) karakter je neophodan, jer niska može biti prefiks druge niske. Na primer, u gore prikazanom trie, niska THE je prefiks niske THERE.

Dakle, osnovna ideja ove strukture je da putanje od korena do listova ili do nekih od unutrašnjih čvorova, kodiraju ključeve, a da se podaci vezani za taj ključ čuvaju u čvoru do kojeg se dolazi pronalaženjem ključa duž putanje. U slučaju niski, koren sadrži praznu reč, a prelaskom preko svake grane se

na do tada formiranu reč nadovezuje još jedan karakter. Pritom, zajednički prefiksi različitih ključeva su predstavljeni istim putanjama od korena do tačke razlikovanja.

U vremenskoj složenosti $O(n)$ možemo odrediti da li trie sadrži string dužine n, jer možemo pratiti lanac koji počinje u korenu. Takodje možemo dodati string dužine n u trie u vremenskoj složenosti $O(n)$ tako sto ćemo najpre pratiti odgovarajući lanac i potom dodati novi čvor u trie (ako je potrebno).

Upotrebom trie strukture podataka, možemo odrediti najduži prefiks datog stringa tako da prefiks pripada datom skupu. Štaviše, kako se u svakom čvoru trie-a skladišti dodatna informacija, možemo izračunati broj stringova koji pripadaju datom skupu i čiji prefiks je data niska.

Trie se može deklarisati kao niz **int trie[N][A];** gde je N maksimalni broj čvorova (maximalna ukupna dužina stringova u skupu) i A je veličina alfabeta nad kojim se grade niske.

Čvorovi trie-a su numerisani $0, 1, 2, \ldots$ tako da broj u korenu ima numeraciju 0, i `trie`$[s][c]$ je naredni čvor u lancu kada se krećemo iz čvora s pomoću karaktera c.

U slučaju da neki čvor ima samo jednog potomka i ne predstavlja kraj nekog ključa, grana do njega i grana od njega se mogu spojiti u jednu, njihovi karakteri nadovezati, a čvor eliminisati. Ovako se dobija kompaktnija reprezentacija prefiksnog drveta.

Svaki trie se može smatrati drvolikim determinističkim konačnim automatom. Postoje radovi u kojima je dat prikaz konačnih jezika koji su generisani trie automatom i svaki trie je kompresovan u rezultujuci deterministicki aciklicki konacan automat.

Pored opšteg asocijativnog pristupa podacima, očigledna primena ove strukture je i implementacija konačnih rečnika, na primer u svrhe automatskog kompletiranja ili provere ispravnosti reči koje korisnik kuca na računaru ili mobilnom telefonu.

Napomenimo još i da ova struktura nije rezervisana za čuvanje stringova. Na primer, u slučaju celih brojeva ili brojeva u pokretnom zarezu, ključ mogu biti niske bitova koje predstavljaju takve brojeve (npr. bitwise trie).

U slučaju konačne azbuke veličine m, složenost operacija u najgorem slučaju je $O(mn)$, gde je n dužina reči koja se traži, umeće ili briše. Pretraga i umetanje se pravolinijski implementiraju, dok je u slučaju brisanja nekada potrebno brisati više od jednog čvora (bottom-up rekurzivno).

Ukoliko su ključevi relativno kratki, prednost prefiksnog drveta je što složenost zavisi od dužine zapisa ključa, a ne od broja elemenata u drvetu. Mana je potreba za čuvanjem pokazivača uz svaki karakter u drvetu.

U nastavku su date implementacije osnovnih operacija sa prefiksnim drvetom na primeru formiranja i pretrage rečnika.

```
#include <iostream>
#include <vector>
#include <string>
#include <unordered_map>
```

```
using namespace std;

// struktura čvora prefiksnog drveta - u svakom čvoru čuvamo niz
// grana obeleženih karakterima ka potomcima i informaciju da li
// je u ovom čvoru kraj neke reči

struct cvor {
  bool krajKljuca = false;
  unordered_map<char, cvor*> grane;
};


// tražimo sufiks reči w koji počinje od pozicije i u drvetu na
// čiji koren ukazuje pokazivač drvo
bool nadji(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan, on je u korenu akko je u korenu obeleženo
  // da je tu kraj reči
  if (i == w.size())
    return drvo->krajKljuca;

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako je nadjemo, rekurzivno tražimo ostatak sufiksa od pozicije i+1
  if(it != drvo->grane.end())
    return nadji(it->second, w, i+1);

  // nismo našli granu sa w[i], pa reč ne postoji
  return false;
}

// tražimo u drvetu na čiji koren ukazuje pokazivač drvo reč w
bool nadji(cvor *drvo, const string& w) {
  return nadji(drvo, w, 0);
}

// umeće sufiks reči w od pozicije i u drvo na čiji koren ukazuje
// pokazivač trie
void umetni(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan samo u korenu beležimo da je tu kraj reči
  if (i == w.size()) {
    drvo->krajKljuca = true;
    return;
  }

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
```

```
  // ako takva grana ne postoji, dodajemo je kreirajući novi čvor
  if(it == drvo->grane.end())
    drvo->grane[w[i]] = new cvor();

  // sada znamo da grana sa w[i] sigurno postoji i preko te grane
  // nastavljamo dodavanje sufiksa koji počinje na i+1;
  umetni(drvo->grane[w[i]], w, i+1);
}

// umeće reč w u drvo na čiji koren ukazuje pokazivač w
void umetni(cvor *drvo, string& w) {
  return umetni(drvo, w, 0);
}

// program kojim testiramo gornje funkcije
int main() {
  cvor* drvo = new cvor();
  vector<string> reci
    {"vas", "vasa", "vasilije", "vatra", "da", "dan", "dud", "duz"};
  vector<string> reci_nema
    {"", "v", "d", "vasiona", "trava", "drvo"};
  for(auto w : reci)
    umetni(drvo, w);
  for(auto w : reci)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  for(auto w : reci_nema)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  return 0;
}
```

Literatura

1. D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*, Addison–Wesley, 1998 (2nd edition).

2. Edward Fredkin. *Trie Memory*, Communications of the ACM. 3 (9): 490–499, 1960

3, Jan Daciuk; Stoyan Mihov; Bruce W. Watson; Richard E. Watson. *Incremental Construction of Minimal Acyclic Finite-State Automata*, Computational Lin-guistics. Association for Computational Linguistics. 26: 3, 2000

4. Ulrich Germann; Eric Joanis; Samuel Larkin. *Tightly packed tries: how to fit large models into memory, and make them load fast, too*, ACL Workshops: Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing , 2009

```
import java.util.HashMap;
import java.util.Map;

//  Trie cvor
class Trie
{
	boolean isLeaf;
	Map<Character, Trie> children;

	// Konstruktor
	Trie() {
		isLeaf = false;
		children = new HashMap<>();
	}

	// Iterativno umetanje stringa  u Trie
	public void insert(String key)
	{
		System.out.println("Umetanje \"" + key + "\"");

		// pocinje se sa korenom
		Trie curr = this;

		// za svaki karakter kljuca
		for (int i = 0; i < key.length(); i++)
		{
			// kreira se novi cvor ako ne postoji grana/put
			if (curr.children.get(key.charAt(i)) == null)
				curr.children.put(key.charAt(i), new Trie());

			// idi do sledeceg cvora
			curr = curr.children.get(key.charAt(i));
		}

		// oznaci da je sledeci cvor list
		curr.isLeaf = true;
	}

	// Iterativna funkcija pretrage kljuca unutarTrie. Vraca  true
	// ako je nadjen ljuc unutar Trie, inace vraca false
	public boolean search(String key)
	{
		System.out.print("Pretraga \"" + key + "\" : ");

		Trie curr = this;

		// za svaki karakter kljuca
		for (int i = 0; i < key.length(); i++)
		{
			// idi do sledeceg cvora
			curr = curr.children.get(key.charAt(i));

			// ako nismo nasli na string na grani  (dostigli smo kraj grane u Trie)
			if (curr == null)
				return false;
		}

		// vrati  true ako tekuci cvor je list i dostigli smo kraj stringa
		return curr.isLeaf;
	}
};
```

```
class TrieImpl
{
	// memorijski efikasna Trie implementacija, koristimo HashMap
	//cuva se samo onaj deo alfabeta koji se koristi, izbegava se koriscenje prostora za null pointer-e
	public static void main (String[] args)
	{
		// konstruisemo novi Trie cvor
		Trie head = new Trie();

		head.insert("vas");
		head.insert("vasa");
		head.insert("vasilije");

		System.out.println(head.search("v"));
		System.out.println(head.search("vasa"));
		System.out.println(head.search("vas"));
		System.out.println(head.search("vasilije"));

		head.insert("v");

		System.out.println(head.search("v"));
		System.out.println(head.search("vasa"));
		System.out.println(head.search("vas"));
		System.out.println(head.search("vasilije"));

	}
}
```

Python 3 rešenje

```
class TrieNode:

	# Trie cvor
	def __init__(self):
		self.grana = [None]*26
		# krajKljuca je True ako cvor predstavlja kraj reci
		self.krajKljuca = False

class Trie:

	# Trie - prefiksno stablo
	def __init__(self):
		self.root = self.dajCvor()

	def dajCvor(self):

		# kreiramo novi cvor stabla trie  (inicijalno prazan cvor)
		return TrieNode()

	def _charToIndex(self,ch):

		# pomocna funkcije konverzije karakter -> indeks
		# konvertujemo u indeks kljuc
		# tj. malo slovo od 'a' do 'z' (oduzimamo ASCII kodove)

		return ord(ch)-ord('a')
```

```
    def insert(self,kljuc):

        # Ubaci kljuc u trie (ako nije prisutan)
        # Ako kljuc je prefix cvora trie-a,
        # samo obelezimo cvor koji je kraj kljuca
        drvo = self.root
        brojKaraktera = len(kljuc)
        for level in range(brojKaraktera):
            index = self._charToIndex(kljuc[level])

            # ako ne postoji tekuci karakter
            if not drvo.grana[index]:
                drvo.grana[index] = self.dajCvor()
            drvo = drvo.grana[index]

        # obelezi poslednji cvor da je list
        drvo.krajKljuca = True

    def search(self, kljuc):

        # nadji kljuc unutar trie
        # vrati  true ako nadjes kljuc
        # inace, vrati false
        drvo = self.root
        brojKaraktera = len(kljuc)
        for level in range(brojKaraktera):
            index = self._charToIndex(kljuc[level])
            if not drvo.grana[index]:
                return False
            drvo = drvo.grana[index]

        return drvo != None and drvo.krajKljuca

# glavni program

# recnik reci koje se sastoje samo od malih slova
recnik = ["vas","vasa","vatra","vasilije","trava",
            "trougao","trapey"]
output = ["nema","ima"]

# instanciramo Trie
t = Trie()

# dodajemo rec u trie
for rec in recnik:
  t.insert(rec)

# trazimo kljuceve unutar trie
print("{} -> {}".format("vas",output[t.search("vas")]))
print("{} -> {}".format("vaske",output[t.search("vaske")]))
print("{} -> {}".format("trougao",output[t.search("trougao")]))
print("{} -> {}".format("trie",output[t.search("trie")]))
```