



Univerzitet u Nišu

Elektronski fakultet

Seminarski rad

Metodi i sistemi za obradu signala

Kompresija podataka primenom adaptivnog Huffman-ovog kodiranja: Vitter-ov metod (Full)

Studenti:

Božidar Mitić 18282

Miloš Jovanović 18196

Profesor:

Prof.dr Miloš Radmanović

Sadržaj

1. Uvod	3
2. Osnove Huffman-ovog kodiranja i dekodiranja.....	3
2.1 Kodiranje.....	3
2.2 Dekodiranje.....	4
2.3 Prosečna Veličina Koda	4
2.4 Visina Huffman stabla.....	5
2.4.1 Najkraće Huffmanovo Stablo.....	5
2.4.2 Najviše Huffmanovo Stablo	5
2.4.3 Primena u Praksi	5
3. Adaptivno Huffman kodiranje.....	6
3.1 Principi Adaptivnog Huffman Kodiranja	6
3.2 Prednosti Adaptivnog Huffman Kodiranja.....	6
3.3 Nedostaci Adaptivnog Huffman Kodiranja.....	7
3.4 Nekompresovani kodovi	7
3.5 Modifikacija Huffman stabla	7
3.5.1 Ažuriranje stabla	8
3.6 Prelivanje brojača.....	8
3.6.1 Uticaj ponovnog skaliranja	9
3.6.2 Kako funkcioniše ponovno skaliranje.....	9
3.7 Prelivanje koda.....	9
3.7.1 Rešenja za prelivanje koda.....	9
3.8 Vitter-ova metoda.....	10
3.8.1 Prednosti Vitter-ovih ideja	10
3.8.2 Specijalna struktura podataka: "floating tree"	10
3.8.3 Performanse poboljšane verzije	10
4. Uputstvo za upotrebu aplikacije	11
5. Kod za kompresiju podataka primenom adaptivnog Huffman-ovog kodiranja: Vitter-ov metod	13
5.1 Klasa Encoder	15
5.2 Klasa Decoder.....	17
5.3 Klasa Node.....	19
5.4 Klasa Tree	20
6. Tabela ulaza i izlaza	24
7. Zaključak.....	24
8. Literatura.....	24

1. Uvod

Kompresija podataka je proces smanjenja količine podataka potrebnih za predstavljanje informacije. To se postiže uklanjanjem redundancije i efikasnijim kodiranjem podataka. Kompresija je korisna jer omogućava brže prenošenje podataka, smanjuje potrebnu memoriju za čuvanje podataka i može smanjiti troškove povezane sa skladištenjem i prenosom. Postoje različite metode kompresije podataka, uključujući sa gubicima (lossy) i bez gubitaka (lossless) metode, koje se koriste zavisno od potreba specifičnih aplikacija.

Jedna od često korišćenih metoda za kompresiju podataka je Huffman kodiranje. Ova metoda je osnova za nekoliko popularnih programa koji se koriste na ličnim računarima. Neki od njih koriste samo Huffman metodu, dok drugi koriste ovu metodu kao jedan korak u višestepenom procesu kompresije. Huffman metoda, koju je razvio D. Huffman 1952. godine, slična je Shannon-Fano metodi, ali obično proizvodi bolje kodove. Kao i Shannon-Fano metoda, Huffman metoda daje najbolje rezultate kada su verovatnoće simbola negativne stepeni broja 2. Glavna razlika između ove dve metode je u tome što Shannon-Fano konstruiše kodove odozgo ka dole (od levog ka desnom bitu), dok Huffman konstruiše stablo kodova odozdo ka gore (gradi kodove od desnog ka levom bitu).

2. Osnove Haffman-ovog kodiranja i dekodiranja

2.1 Kodiranje

Proces počinje izradom liste svih simbola abecede poređanih po opadajućoj verovatnoći. Zatim se konstruiše stablo, sa simbolom na svakom listu, od dole ka gore. To se radi u koracima, pri čemu se u svakom koraku biraju dva simbola sa najmanjim verovatnoćama, dodaju se na vrh delimičnog stabla, brišu se sa liste i zamenjuju pomoćnim simbolom koji predstavlja oba. Kada se lista smanji na samo jedan pomoćni simbol (koji predstavlja celu abecedu), stablo je kompletno. Stablo se zatim pretražuje kako bi se odredili kodovi simbola.

Primer

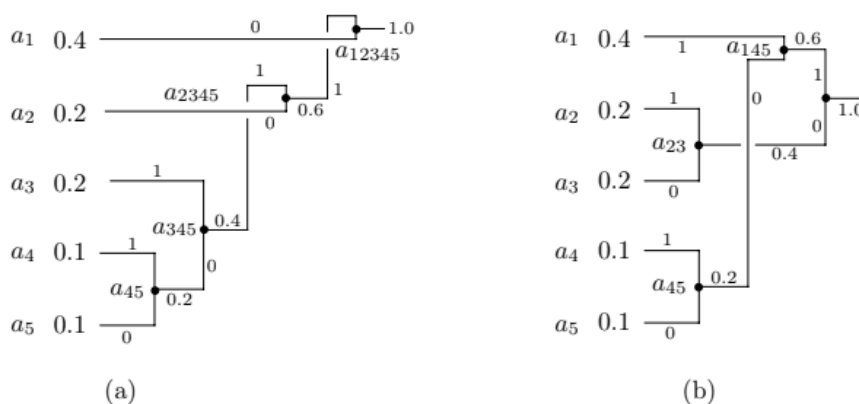


Figure 2.14: Huffman Codes.

2.2 Dekodiranje

Pre nego što započne kompresiju data stream-a, kompresor (enkoder) mora da odredi kodove. To čini na osnovu verovatnoća (ili frekvencija pojavljivanja) simbola. Verovatnoće ili frekvencije moraju biti prisutne u komprimovanom toku, kako bi bilo koji Huffman dekompresor (dekoder) mogao da dekompresuje tok. Ovo je jednostavno, jer su frekvencije celi brojevi, a verovatnoće se mogu zapisati kao skalirani celi brojevi. To obično dodaje samo nekoliko stotina bajtova komprimovanom toku. Takođe je moguće zapisati kodove promenljive veličine direktno u tok, ali to može biti nezgodno, jer kodovi imaju različite veličine. Takođe je moguće zapisati Huffman stablo u tok, ali to može biti duže nego samo frekvencije.

U svakom slučaju, dekodeer mora znati šta se nalazi na početku toka, pročitati to i konstruisati Huffman stablo za abecedu. Tek tada može čitati i dekodirati ostatak toka. Algoritam za dekodiranje je jednostavan. Počinje se od korena i čita se prvi bit iz komprimovanog toka. Ako je nula, ide se prema donjem rubu stabla; ako je jedan, ide se prema gornjem rubu. Čita se sledeći bit i prelazi još jedan rub ka listovima stabla. Kada dekodeer stigne do lista, pronalazi originalni, nekomprimovani kod simbola (obično njegov ASCII kod), i taj kod dekodeer emituje. Proces zatim ponovo počinje od korena sa sledećim bitom.

2.3 Prosečna Veličina Koda

Prosečna veličina koda je važan koncept u teoriji kompresije podataka, posebno kada se koristi Huffman kodiranje. Ona predstavlja prosečan broj bitova potrebnih za kodiranje simbola u datom skupu podataka. Efikasnost Huffman kodiranja može se direktno oceniti kroz ovu meru. Pogledajmo primer sa pet simbola i njihovim verovatnoćama, zajedno sa odgovarajućim Huffman kodovima:

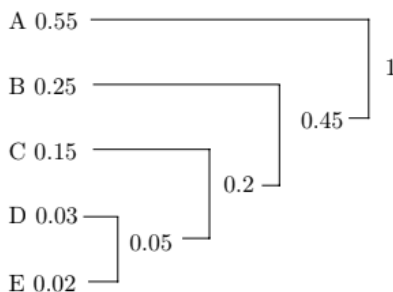
- A: 0.55 (1-bitni kod)
- B: 0.25 (2-bitni kod)
- C: 0.15 (3-bitni kod)
- D: 0.03 (4-bitni kod)
- E: 0.02 (4-bitni kod)

Prosečna veličina koda se izračunava kao:

Prosečna veličina koda = $0.55 \times 1 + 0.25 \times 2 + 0.15 \times 3 + 0.03 \times 4 + 0.02 \times 4 = 1.7$ bita po simbolu

Ovaj rezultat može se dobiti i sabiranjem vrednosti internih čvorova u Huffman stablu:

$$0.05 + 0.2 + 0.45 + 1 = 1.70. 0.05 + 0.2 + 0.45 + 1 = 1.70. 0.05 + 0.2 + 0.45 + 1 = 1.7$$



2.4 Visina Huffman stabla

Visina Huffmanovog stabla je važan parametar jer predstavlja dužinu najdužeg koda u stablu. Ovo može biti bitno u nekim primenama kompresije podataka. Na primer, Deflate metod ograničava dužine određenih Huffmanovih kodova na samo tri bita.

2.4.1 Najkraće Huffmanovo Stablo

Najkraće Huffmanovo stablo se dobija kada simboli imaju jednake verovatnoće. U ovom slučaju, simboli se kombinuju u parove na najnižem nivou stabla, što rezultuje uravnoteženim ili gotovo uravnoteženim stablom visine $\log_2 n$, gde je n broj simbola. Kada je broj simbola stepen broja 2, visina stabla je tačno $\log_2 n$.

2.4.2 Najviše Huffmanovo Stablo

Da bismo generisali najviše stablo, verovatnoće simbola moraju biti takve da svaka iteracija Huffmanovog algoritma povećava visinu stabla za 1. Prvi korak kombinuje dva simbola sa najmanjim verovatnoćama a i b , formirajući čvor sa verovatnoćom $a + b$. Svaki naredni korak kombinuje rezultat prethodnog koraka sa jednim od preostalih simbola, čime se visina stabla povećava.

Simboli moraju imati verovatnoće koje formiraju Fibonaccijev niz: $p_1=a, p_2=b, p_3=a+b, p_4=b+(a+b), p_5=(a+b)+(a+2b), p_1=a, p_2=b, p_3=a+b, p_4=b+(a+b), p_5=(a+b)+(a+2b)$, itd. Ove verovatnoće formiraju Fibonaccijev niz čiji su prvi elementi a i b . Na primer, za $a = 5$ i $b = 2$, niz je 5, 2, 7, 9, 16, što rezultira stablom maksimalne visine 4.

2.4.3 Primena u Praksi

U praksi, verovatnoće simbola se izračunavaju brojanjem njihovih pojavljivanja u ulaznom fajlu. Da bi ulazni fajl rezultirao maksimalno visokim Huffmanovim stablom, frekvencije pojavljivanja simbola moraju formirati Fibonaccijev niz. Na primer, za fajl sa devet simbola A–I, frekvencije moraju biti 1, 1, 2, 3, 5, 8, 13, 21, 34. Ukupan zbir ovih frekvencija je 88, pa fajl mora biti bar te dužine da bi se dobilo stablo visine 8.

Ako set simbola ima Fibonaccijeve verovatnoće i rezultira stablom sa kodovima koji su predugi, stablo se može preoblikovati blago modifikovanjem verovatnoća simbola tako da ne formiraju tačan Fibonaccijev niz.

3. Adaptivno Huffman kodiranje

Adaptivno Huffman kodiranje je dinamička tehnika kompresije podataka koja prilagođava Huffmanovo stablo u hodu, dok se simboli obrađuju. Ovo kodiranje je posebno korisno kada frekvencije simbola nisu poznate unapred, jer omogućava kompresiju i dekompresiju u realnom vremenu, prilagođavajući se trenutnom nizu simbola. Metoda je zasnovana na radu Newtona Fallera, Roberta Gallagera i Donalda Knutha

3.1 Principi Adaptivnog Huffman Kodiranja

- 1. Inicijalizacija:** Proces započinje sa praznim Huffmanovim stablom. Nijedan simbol nema dodeljen kod na početku.
- 2. Kodiranje i Dekodiranje u Hodu:**
 - **Kodiranje:** Kada se pročita novi simbol, proverava se da li je već prisutan u stablu.
 - Ako nije prisutan, simbol se dodaje u stablo, dodeljuje mu se kod, i emitira se u nekompresovanom obliku, zajedno sa specijalnim escape kodom koji označava da je u pitanju novi simbol.
 - Ako je simbol već prisutan, emitira se njegov trenutni kod i povećava se njegova frekvencija, što može zahtevati reorganizaciju stabla.
 - **Dekodiranje:** Dekoder koristi isto početno prazno stablo i prati iste korake kao i kodiranje.
 - Kada naiđe na escape kod, dekodeer zna da sledeći niz bitova predstavlja novi simbol koji još nije bio kodiran, te ga dodaje u stablo.
 - Za svaki poznati kod, dekodeer koristi trenutno stablo da dešifruje simbol i ažurira frekvencije.
- 3. Escape Kod:** Escape kod je specijalni kod koji označava da sledeći simbol u nizu nije još bio obrađen i pojavljuje se prvi put. Ovaj kod je varijabilne dužine i menja se kako se stablo ažurira.
- 4. Reorganizacija Stabla:** Svaki put kada se frekvencija simbola promeni, stablo se reorganizuje kako bi ostalo optimalno (najkraći kodovi za najčešće simbole). To može uključivati premještanje čvorova unutar stabla kako bi se održao optimalan raspored.

3.2 Prednosti Adaptivnog Huffman Kodiranja

- **Bez Prethodnog Znanja:** Ne zahteva prethodno znanje o distribuciji simbola, što ga čini idealnim za nepredvidive ili dinamične skupove podataka.
- **Realno-vremenska Kompresija:** Omogućava kompresiju podataka dok se unose, bez potrebe za dvostrukim prolazom kroz podatke kao kod semiadaptivnog kodiranja.
- **Efikasna Upotreba Memorije:** Početno stablo je prazno, čime se smanjuju memorijski zahtevi u poređenju sa statičkim metodama koje zahtevaju čuvanje cele kodne tabele.

3.3 Nedostaci Adaptivnog Huffman Kodiranja

- **Kompleksnost Implementacije:** Ažuriranje stabla i održavanje njegovog balansa u realnom vremenu može biti složeno.
- **Brzina:** Proces ažuriranja stabla može biti sporiji u poređenju sa statičkim metodama kodiranja, posebno za velike skupove podataka.

Escape kod

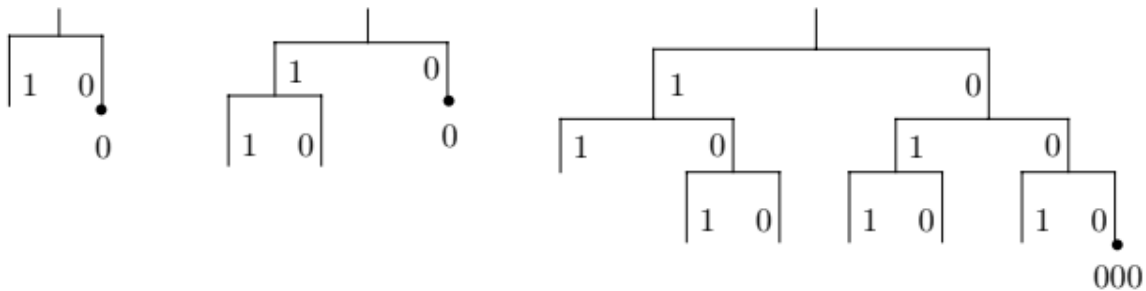


Figure 2.31: The Escape Code.

3.4 Nekompresovani kodovi

Ako su simboli koji se kompresuju ASCII karakteri, jednostavno im se mogu dodeliti njihovi ASCII kodovi kao nekompresovani kodovi. U opštem slučaju, gde simboli mogu biti bilo koji, nekompresovani kodovi različitih veličina mogu biti dodeljeni jednostavnom metodom.

Evo primera za slučaj gde je $n = 24$. Prvih 16 simbola može biti dodeljeno brojevima od 0 do 15 kao njihovi kodovi. Ovi brojevi zahtevaju samo 4 bita, ali ih kodiramo u 5 bita. Simboli od 17 do 24 mogu biti dodeljeni brojevima $17-16-1 = 0$, $18-16-1 = 1$ do $24-16-1 = 7$ kao 4-bitni brojevi. Na kraju imamo šesnaest 5-bitnih kodova 00000, 00001, ..., 01111, praćenih sa osam 4-bitnih kodova 0000, 0001, ..., 0111.

Generalno, pretpostavljamo alfabet koji se sastoji od n simbola a_1, a_2, \dots, a_n . Biramo cele brojeve m i r tako da važi $2^m \leq n < 2^{m+1}$ i $r = n - 2^m$. Prvih 2^m simbola se kodira kao $(m + 1)$ -bitni brojevi od 0 do $2^m - 1$. Preostali simboli se kodiraju kao m -bitni brojevi tako da je kod za a_k jednak $k - 2^m - 1$. Ovaj kod se takođe naziva faznim binarnim kodom

3.5 Modifikacija Huffman stabla

Glavna ideja je da se stablo proverava svaki put kada se unese novi simbol. Ako stablo više nije Huffmanovo stablo, treba ga ažurirati. Pogled na Sliku 2.32a pokazuje šta znači biti Huffmanovo stablo. Stablo na slici sadrži pet simbola: A, B, C, D i E. Prikazano je sa simbolima

i njihovim frekvencijama (u zgradama) nakon što je obrađeno 16 simbola. Osobina koja čini stablo Huffmanovim jeste ta da ako ga skeniramo nivo po nivo, s leva na desno i odozdo prema gore, frekvencije će biti u ne-silaznom poretku. Donji levi čvor (A) ima najnižu frekvenciju, a gornji desni čvor (korijen) ima najvišu frekvenciju. Ovo se naziva osobinom srodnika (sibling property).

Simbol sa visokom frekvencijom pojavljivanja treba da ima kraći kod. Zato mora biti visoko u stablu. Zahtjev da frekvencije budu poredane s leva na desno na svakom nivou nije neophodan, ali pojednostavljuje proces ažuriranja stabla.

3.5.1 Ažuriranje stabla

Evo sažetka operacija potrebnih za ažuriranje stabla:

1. Poređenje čvora X sa naslednicima:

- Ako neposredni naslednik ima frekvenciju $F + 1$ ili veću, redosled je u redu i nema potrebe za promenom.
- Ako neki naslednici čvora X imaju identične frekvencije F ili manje, X se menja sa poslednjim čvorom u toj grupi (osim sa svojim roditeljem).

2. Povećanje frekvencije X:

- Povećaj frekvenciju čvora X sa F na $F + 1$.
- Povećaj frekvencije svih njegovih roditelja.

3. Ponavljanje petlje:

- Ako je X koren, petlja se zaustavlja; inače, petlja se ponavlja sa roditeljem čvora X.

Ovaj algoritam osigurava da stablo ostane Huffmanovo stablo i da simboli sa većim frekvencijama imaju kraće kodove, što povećava efikasnost kompresije.

Važno je napomenuti da nakon reskaliranja brojeva, novi simboli koji se čitaju i kompresuju imaju veći uticaj na brojeve nego stari simboli (oni koji su brojani pre reskaliranja). Ovo se pokazuje kao korisno jer iskustvo pokazuje da verovatnoća pojave simbola više zavisi od simbola koji su se pojavili neposredno pre njega nego od simbola koji su se pojavili u dalekoj prošlosti

3.6 Prelivanje brojača

Brojači frekvencija se akumuliraju u Huffmanovom stablu u poljima fiksne veličine, i ta polja mogu doći do preliva. Na primer, 16-bitno bez predznaka polje može akumulirati brojeve do $2^{16} - 1 = 65,535$. Jednostavno rešenje je pratiti polje brojača korena svaki put kada se poveća, i kada dostigne maksimalnu vrednost, ponovo skalirati sve frekvencije deljenjem sa 2 (celo brojno deljenje). U praksi, ovo se radi tako što se prvo skaliraju polja brojača listova, a zatim se ažuriraju brojevi unutrašnjih čvorova. Svaki unutrašnji čvor dobija zbir brojeva svojih dece. Problem je što su brojevi celi, i celo brojno deljenje smanjuje preciznost. Ovo može promeniti Huffmanovo stablo tako da više ne zadovoljava osobinu srodnika.

3.6.1 Uticaj ponovnog skaliranja

Treba napomenuti da nakon ponovnog skaliranja brojača, novi simboli koji se čitaju i kompresuju imaju veći uticaj na brojeve nego stari simboli (oni koji su brojani pre ponovnog skaliranja). Ovo se pokazuje korisnim jer iskustvo pokazuje da verovatnoća pojave simbola više zavisi od simbola koji su se pojavili neposredno pre njega nego od simbola koji su se pojavili u dalekoj prošlosti.

3.6.2 Kako funkcioniše ponovno skaliranje

1. Praćenje brojača korena:

- Polje brojača korena se prati svaki put kada se poveća. Kada dostigne maksimalnu vrednost (npr. 65,535 za 16-bitni brojač), pokreće se ponovno skaliranje.

2. Ponovno skaliranje frekvencija:

- Svi brojači frekvencija se dele sa 2 (celo brojno deljenje).
- Prvo se skaliraju polja brojača listova, a zatim se ažuriraju brojevi unutrašnjih čvorova tako da svaki unutrašnji čvor dobija zbir brojeva svojih dece.

3. Uticaj na stablo:

- Ponovno skaliranje može smanjiti preciznost brojeva zbog celog brojnog deljenja, što može promeniti strukturu stabla i narušiti osobinu srodnika (sibling property).

3.7 Prelivanje koda

Još ozbiljniji problem je prelivanje koda. Ovo se može desiti kada se mnogo simbola doda stablu i ono postane visoko. Kodovi se ne skladište direktno u stablu jer se stalno menjaju, tako da kompresor mora svaki put da izračuna kod za simbol X kada se X unese. Evo detalja:

1. Pronalaženje simbola X u stablu:

- Enkoder mora da locira simbol X u stablu. Stablo treba da bude implementirano kao niz struktura, gde je svaka struktura čvor, a niz se pretražuje linearno.

2. Ako X nije pronađen:

- Izlazi escape kod, praćen nekompresovanim kodom simbola X. Zatim se X dodaje stablu.

3. Ako je X pronađen:

- Kompresor se kreće od čvora X ka korenu, gradeći kod bit po bit. Svaki put kada se ide od levog deteta ka roditelju, dodaje se "1" u kod. Kretanje od desnog deteta ka roditelju dodaje "0" (ili obrnuto, ali ovo treba da bude konzistentno jer dekodirer to isto radi).
- Ti bitovi moraju biti negde sačuvani jer se moraju emitovati obrnutim redosledom. Kada stablo postane više, kodovi postaju duži. Ako se akumuliraju u 16-bitnom celobrojnom tipu, kodovi duži od 16 bitova izazvaće kvar.

3.7.1 Rešenja za prelivanje koda

1. Lista povezanih čvorova:

- Bitovi koda mogu se akumulirati u povezanoj listi, gde se mogu kreirati novi čvorovi, ograničeni samo dostupnom memorijom. Ovo je generalno rešenje, ali je sporo.

2. Velika celobrojna varijabla:

- Kodovi se mogu akumulirati u velikoj celobrojnoj varijabli (na primer, 50-bitna) i dokumentovati maksimalnu veličinu koda od 50 bitova kao jedno od ograničenja programa.

Srećom, ovaj problem ne utiče na proces dekodiranja. Dekoder čita kompresovani kod bit po bit i koristi svaki bit da se kreće levo ili desno niz stablo dok ne dođe do listnog čvora. Ako je listni čvor escape kod, dekodeer čita nekompresovani kod simbola sa kompresovanog toka (i dodaje simbol stablu). U suprotnom, nekompresovani kod se nalazi u listnom čvoru.

3.8 Vitter-ova metoda

Poboljšanje originalnog algoritma, koje je predložio Vitter, uključuje sledeće ključne ideje:

1. Implicitno numerisanje čvorova:

- Umesto dosadašnjeg načina numerisanja, koristi se shema koja numerise čvorove odozdo prema gore i na svakom nivou s leva na desno.

2. Ažuriranje Huffmanovog stabla:

- Stablo se ažurira tako da važi sledeća invarijanta: za svaku težinu w , svi listovi težine w prethode (u smislu implicitnog numerisanja) svim unutrašnjim čvorovima iste težine.

3.8.1 Prednosti Vitter-ovih ideja

1. Stabilnost nivoa čvorova:

- U originalnom algoritmu, moguće je da preuređenje stabla spusti čvor na jedan nivo niže. U poboljšanoj verziji, to se ne dešava.

2. Ograničeno pomeranje čvorova:

- U originalnom algoritmu, prilikom ažuriranja Huffmanovog stabla, neki čvorovi mogu biti pomereni naviše. U poboljšanoj verziji, najviše jedan čvor može biti pomeren naviše.

3. Optimizacija visine stabla:

- Poboljšana verzija minimizuje zbir udaljenosti od korena do listova i ima minimalnu visinu stabla.

3.8.2 Specijalna struktura podataka: "floating tree"

Koristi se specijalna struktura podataka nazvana "floating tree" da bi se olakšalo održavanje potrebne invarijante.

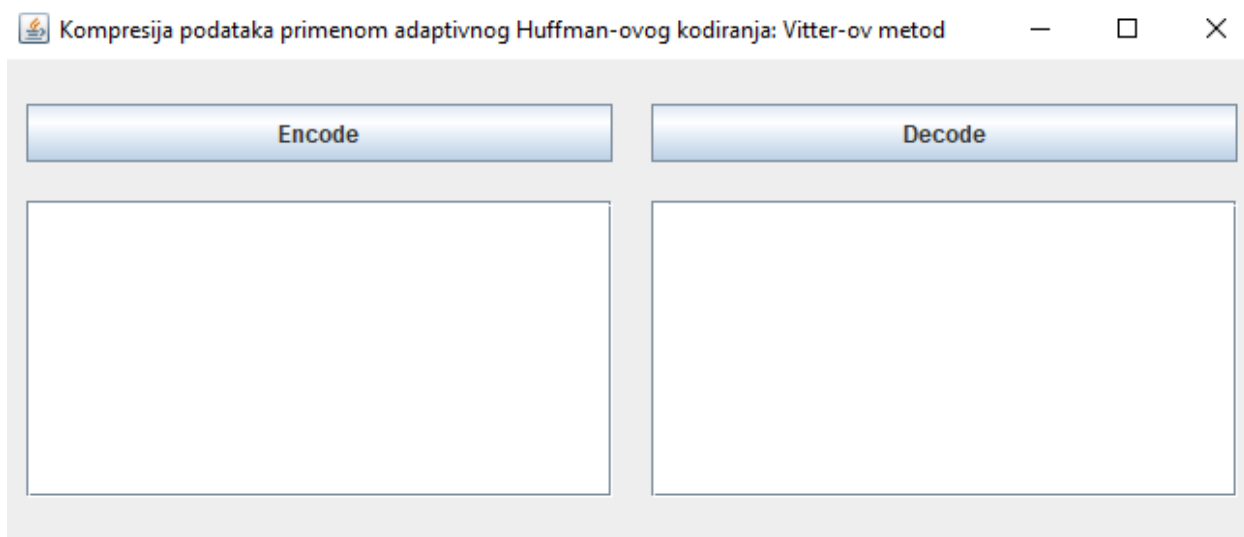
3.8.3 Performanse poboljšane verzije

Pokazano je da ova verzija značajno nadmašuje originalni algoritam. Konkretno, ako metoda Huffmanovog kodiranja u dva prolaza kompresuje ulazni fajl od n simbola na S bitova, originalni adaptivni Huffmanov algoritam može ga kompresovati na najviše $2S+n$ bitova, dok poboljšana verzija može ga kompresovati na $S+n$ bitova što je značajna razlika! Ovi rezultati ne zavise od veličine alfabeta, već samo od veličine nnn podataka koji se kompresuju i njihove prirode (koja određuje S).

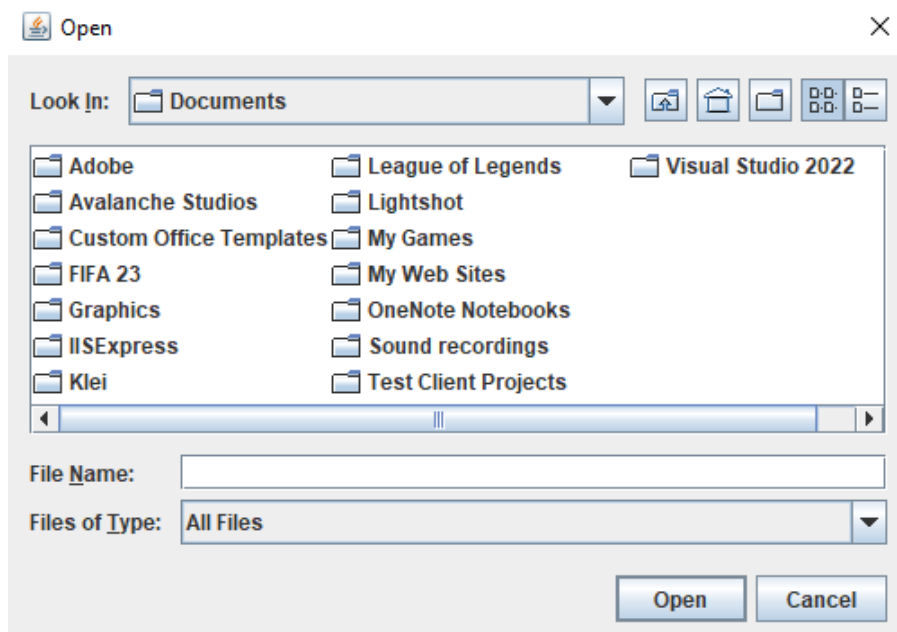
4. Uputstvo za upotrebu aplikacije

Ovaj projekat predstavlja alat za kompresiju i dekompresiju različitih tipova fajlova korišćenjem. Adaptivnog Huffman-ovog algoritma: Vitter-ov metod. Projekat je napisan na programskom jeziku Java. Program koristi standardne Java kolekcije kao i ulazne i izlazne biblioteke kao i JavaX Swing biblioteku za GUI.

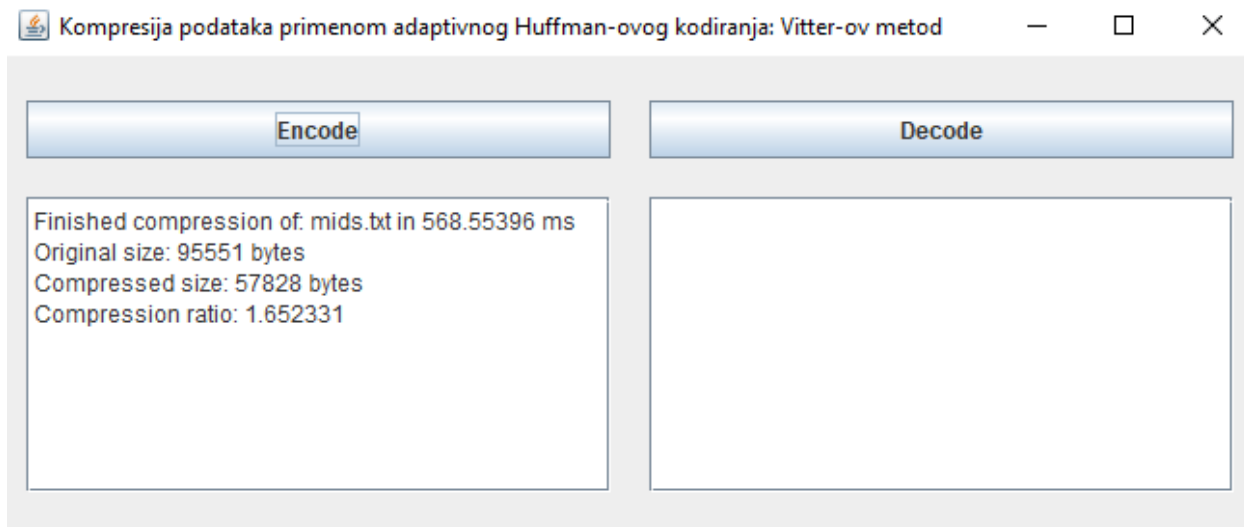
Za korišćenje aplikacije potrebno je imati instaliranu najnoviju verziju Jave i JDK-a. (<https://www.oracle.com/java/technologies/javase/jdk17-archive-downloads.html>).



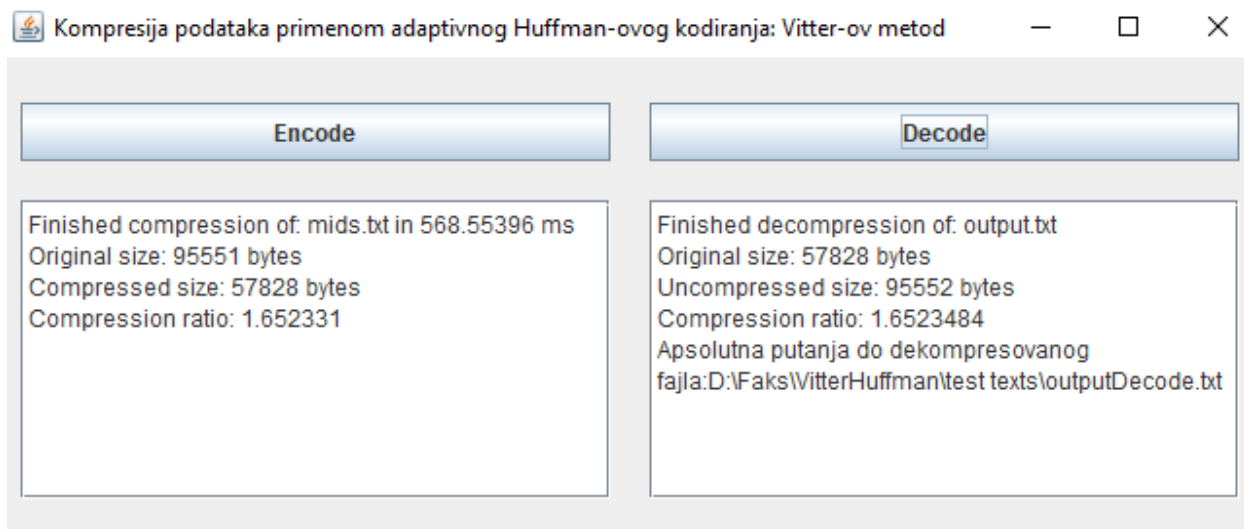
Korisnik najpre pritiskom na dugme “Encode” otvara file system u kome bira fajl koji želi da kompresuje.







Korisnik bira fajl i pritiskom na dugme “Open” pokreće proces kompresije.



Nakon kompresije, na ekranu se ispisuju informacije o njoj i kompresovana verzija fajla se nalazi u direktorijumu projekta. Zatim klikom na dugme “Decode” pokreće se dekompresija poslednjeg kompresovanog fajla(fajl koji se nalazi u direktorijumu projekta). Nakon čega se ispisuju informacije o dekompresiji.



Nakon dekompresije, dekompresovani fajl nalazi se u direktorijumu projekta i zove se “outputDecode.(odgovarajuća ekstenzija)”

	outputDecode	6/28/2024 10:07 PM	Microsoft Word D...	929 KB
	outputDecode	6/28/2024 10:07 PM	JPG File	182 KB
	outputDecode	6/28/2024 10:07 PM	MP3 File	6,321 KB
	outputDecode	6/28/2024 10:38 PM	Text Document	94 KB

5. Kod za kompresiju podataka primenom adaptivnog Huffman-ovog kodiranja: Vitter-ov metod

Najpre se pokreće main metoda iz klase mainApp, u kojoj se izvršavaju potreba podešavanja GUI-a

```
public class mainApp {
    public static void main(String[] args){
        basicForm form = new basicForm();
        JFrame frame = new JFrame("Kompresija podataka primenom adaptivnog Huffman-ovog kodiranja: Vitter-ov metod");
        frame.setContentPane(form.getPanel());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.setVisible(true);
    }
}
```

U konstruktoru za formu (GUI) izvršava se sledeći kod koji postavlja ActionListener za dugme Encode. Kada korisnik klikne na dugme, otvara se dijalog za izbor fajla pomoću JFileChooser. Nakon što korisnik izabere fajl, kod dobija apsolutnu putanju izabranog fajla i izvlači njegovu ekstenziju. Ekstenzija se zatim piše u fajl test texts/ekstenzija.txt, a ako dođe do greške prilikom pisanja, prikazuje se poruka o grešci.

Nakon toga, kreira se objekat Encoder sa putanjama ulaznog i izlaznog fajla, kao i novo Huffman-ovo stablo (Tree). Merenje vremena se započinje pre kodiranja koristeći System.nanoTime(), zatim se vrši kodiranje fajla pomoću encoder.encode(tree), i na kraju se zabeležava vreme završetka kodiranja. Rezultati kompresije, uključujući ime ulaznog fajla, vreme trajanja kompresije, originalnu veličinu fajla, kompresovanu veličinu fajla i kompresioni odnos, prikazuju se u encodeInfoPane. Ako korisnik ne izabere nijedan fajl, prikazuje se poruka "No file selected" u encodeInfoPane.

```

encodeButton.addActionListener(new ActionListener() { // Milijush
    @Override // Milijush
    public void actionPerformed(ActionEvent e) {
        JFileChooser fileChooser = new JFileChooser();
        int result = fileChooser.showOpenDialog( parent: null);
        if (result == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            String absolutePath = selectedFile.getAbsolutePath();
            String outputFilePath = "test texts/output.txt"; // Adjust output file path as needed
            File file = new File(absolutePath);
            String extension = getFileExtension(file);
            String newExtension = "." + extension;
            String filePath = "test texts/ekstenzija.txt";

            try (BufferedWriter writer = new BufferedWriter(new FileWriter(filePath))) {
                writer.write(newExtension);
                encodeInfoPane.setText("Successfully written extension to file: " + filePath + "\n");
            } catch (IOException ex) {
                encodeInfoPane.setText("Error writing to file " + filePath + ": " + ex.getMessage() + "\n");
            }

            Encoder encoder = new Encoder(absolutePath, outputFilePath);
            Tree tree = new Tree();
            File in = new File(absolutePath);
            long startTime = System.nanoTime();
            encoder.encode(tree);
            long endTime = System.nanoTime();
            File out = new File(outputFilePath);

            // Example output
            StringBuilder sb = new StringBuilder();
            sb.append("Finished compression of: ").append(in.getName()).append(" in ").append((float) (endTime - startTime) / 1000000).append(" ms\n");
            sb.append("Original size: ").append(in.length()).append(" bytes\n");
            sb.append("Compressed size: ").append(out.length()).append(" bytes\n");
            sb.append("Compression ratio: ").append((float) in.length() / (float) out.length()).append("\n");
            encodeInfoPane.setText(sb.toString());
        } else {
            encodeInfoPane.setText("No file selected\n");
        }
    }
});

```

U konstruktoru se takođe nalazi i deo koda koji postavlja ActionListener na dugme Decode. Kada korisnik klikne na dugme, prvo se čitaju svi bajtovi iz fajla test texts/ekstenzija.txt i konvertuju u string kako bi se dobila ekstenzija originalnog fajla. Kreira se objekat Decoder sa putanjama ulaznog i izlaznog fajla, kao i novo Huffman-ovo stablo (Tree). Proces dekodiranja se vrši pomoću metode `dec.decode(tree)`. Nakon dekodiranja, dobijeni fajl se proverava da li ima željenu ekstenziju.

Ako trenutna ekstenzija fajla nije ista kao željena ekstenzija, menja se ekstenzija fajla. Ako već postoji fajl sa željenim imenom, taj fajl se briše. Zatim se pokušava promeniti ime izlaznog fajla. Ako je promena uspešna, ispisuju se informacije o završetku dekompresije, uključujući apsolutnu putanju do dekompresovanog fajla. Ako promena imena nije uspešna, prikazuje se poruka o grešci. U slučaju da fajl već ima željenu ekstenziju, ispisuju se osnovne informacije o dekompresiji i apsolutna putanja do fajla. Sve informacije se prikazuju u `decodeInfoPane`. Ako dođe do greške tokom dekompresije, prikazuje se poruka o grešci u `decodeInfoPane`.

```

decodeButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            // Read all bytes from the file and convert to string
            String ekstenzija = new String(Files.readAllBytes(Paths.get("test texts/ekstenzija.txt")));
            Decoder dec = new Decoder("test texts/output.txt", "test texts/outputDecode.txt");
            Tree tree = new Tree();
            File in = new File("test texts/output.txt");
            dec.decode(tree);
            File out = new File("test texts/outputDecode.txt");
            long duzina=out.length();

            String currentFileName = out.getName();
            String currentExtension = currentFileName.substring(currentFileName.lastIndexOf("."));
            String desiredExtension = ekstenzija;

            StringBuilder sb = new StringBuilder();
            if (!currentExtension.equals(desiredExtension)) {
                String fileNameWithNewExtension = currentFileName.replaceFirst("\\.\\w+$", "." + desiredExtension);
                File renamedFile = new File(out.getParent(), fileNameWithNewExtension);
                if (renamedFile.exists()) {
                    if (renamedFile.delete()) {
                        System.out.println("Deleted existing file with the same name.\n");
                    } else {
                        System.out.println("Failed to delete existing file.\n");
                    }
                }
                if (out.renameTo(renamedFile)) {
                    System.out.println("File extension changed successfully.\n");
                    String absolutePathFile = renamedFile.getAbsolutePath();
                    sb.append("Finished decompression of: ").append(in.getName()).append("\n");
                    sb.append("Original size: ").append(in.length()).append(" bytes\n");
                    sb.append("Uncompressed size: ").append(duzina).append(" bytes\n");
                    sb.append("Compression ratio: ").append((float)duzina/(float)in.length()).append("\n");
                    sb.append("Apsolutna putanja do dekompresovanog fajla:").append(absolutePathFile);
                } else {
                    System.out.println("Failed to change file extension.\n");
                }
            }
        }
    }
});

```

```

        } else {
            System.out.println("File extension is already " + desiredExtension);
            String absolutePathFile = out.getAbsolutePath();
            sb.append("Finished decompression of: ").append(in.getName()).append("\n");
            sb.append("Original size: ").append(in.length()).append(" bytes\n");
            sb.append("Uncompressed size: ").append(duzina).append(" bytes\n");
            sb.append("Compression ratio: ").append((float)duzina/(float)in.length()).append("\n");
            sb.append("Apsolutna putanja do dekompresovanog fajla:").append(absolutePathFile);
        }

        decodeInfoPane.setText(sb.toString());
    } catch (IOException exec) {
        decodeInfoPane.setText("Error during decompression: " + exec.getMessage() + "\n");
    }
}
});
}
}

```

5.1 Klasa Encoder

Polja:

- public FileInputStream in = null;; Ovaj objekat se koristi za čitanje bajtova iz ulaznog fajla.
- public BitByteOutputStream out = null;; Ovaj objekat se koristi za pisanje enkodiranih bajtova u izlazni fajl.

Metode:

1. `public static String getFileExtension(File file):`
 - Ova metoda vraća ekstenziju datog fajla.
 - Prima objekat `File` kao argument.
 - Vraća deo imena fajla koji dolazi nakon poslednje tačke (.). Ako tačka nije pronađena, vraća prazan string.
2. (Konstruktor) `public Encoder(String in, String out):`
 - Konstruktor klase `Encoder`.
 - Prima putanje do ulaznog i izlaznog fajla kao stringove.
 - Otvara ulazni fajl za čitanje i izlazni fajl za pisanje. Ako fajlovi ne mogu biti otvoreni, ispisuje se stack trace greške.
3. `public void encode(Tree tree):`
 - Ova metoda vrši enkodiranje podataka iz ulaznog fajla koristeći Huffmanovo stablo.

```
public class Encoder { 4 usages  ↗ Miljush

    public FileInputStream in = null; 4 usages
    public BitOutputStream out = null; 7 usages
    public static String getFileExtension(File file) {...}
    public Encoder(String in, String out) { 1 usage  ↗ Miljush
        System.out.println(in);
        try {...} catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }

    public void encode(Tree tree) {...}
```

Metoda `encode` enkodira podatke iz ulaznog fajla koristeći Huffmanovo stablo. Ona čita bajt po bajt iz ulaznog fajla i za svaki karakter kreira privremeni buffer za skladištenje bitova. Proverava da li stablo sadrži karakter, a ako ga sadrži, dobija Huffmanov kod za taj karakter i upisuje ga bit po bit u izlazni fajl. Ako stablo ne sadrži karakter, dobija kod za karakter, upisuje ga bit po bit u izlazni fajl, zatim upisuje karakter kao bajt u izlazni fajl i ubacuje karakter u stablo.


```

public void encode(Tree tree) { 1 usage  Miljush *
    try {
        int c = 0;
        while((c = in.read()) != -1) {
            ArrayList<Boolean> buffer = new ArrayList<>();
            if (tree.contains(c)) {

                int len = tree.getCode(c, seen: true, buffer);
                for(len=len-1 ;len>=0;len--){
                    out.writeBit(buffer.get(len));
                }
                tree.insertInto((int)c);
            }

            else {
                int len = tree.getCode(c, seen: false, buffer);
                for(len=len-1 ;len>=0;len--){
                    out.writeBit(buffer.get(len));
                }
                out.writeByte(c);
                tree.insertInto(c);
            }
        }
        out.flush();
    }
    catch (IOException e) {
        System.err.println("Error reading from input");
        e.printStackTrace();
    }
    finally {
        if(in != null) {
            try {
                in.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if(out != null) {
            out.close();
        }
    }
}

```

5.2 Klasa Decoder

Polja:

1. **private BitInputStream in = null;**
 - Ovaj objekat se koristi za čitanje bitova iz ulaznog fajla.
2. **private FileOutputStream out = null;**

- Ovaj objekat se koristi za pisanje dekodiranih bajtova u izlazni fajl.

Metode:

1. (Konstruktor) **public Decoder(String in, String out):**

- Konstruktor klase Decoder.
- Prima putanje do ulaznog i izlaznog fajla kao stringove.
- Otvara ulazni fajl za čitanje bitova i izlazni fajl za pisanje bajtova. Ako fajlovi ne mogu biti otvoreni, ispisuje se stack trace greške.

2. **public void decode(Tree tree):**

- Ova metoda vrši dekodiranje podataka iz ulaznog fajla koristeći Huffmanovo stablo.

3. **private int readByte(BitInputStream in) throws IOException:**

- Pomoćna metoda koja čita jedan bajt (8 bitova) iz ulaznog toka bitova.
- Koristi petlju da pročita 8 bitova iz ulaznog toka i skladišti ih u bitBuffer.
- Vraća vrednost bitBuffer koja predstavlja jedan bajt.

```
public class Decoder { 3 usages  ↗ Miljush

    private BitInputStream in = null; 6 usages
    private FileOutputStream out = null; 6 usages

    public Decoder(String in, String out) {...}

    public void decode(Tree tree) {...}

    private int readByte(BitInputStream in) throws IOException {...}

}
```

Metoda decode dekodira podatke iz ulaznog fajla koristeći Huffmanovo stablo. Prvo se proverava da li je stablo prazno, u kom slučaju se čita prvi bajt iz ulaznog toka i upisuje direktno u izlazni fajl. Nakon toga, svaki bit iz ulaznog toka se čita i koristi za navigaciju kroz stablo: ako je bit 1, prelazi se na desno dete čvora, a ako je 0, na levo dete. Kada se stigne do lista ili specijalnog NYT čvora, odgovarajuća vrednost se piše u izlazni fajl i ubacuje u stablo. Svi ulazni i izlazni tokovi se zatvaraju na kraju operacije, a greške tokom procesa se ispisuju kako bi korisnik bio obavešten o eventualnim problemima.

```

public void decode(Tree tree) { 1 usage  ± Miljush *
    try {
        int c = 0;
        if(tree.isEmpty()) { // Samo ispise prvi bajt
            int bitBuffer = 0;
            for(int i = 0; i<8;i++) {
                c = in.read();
                bitBuffer |= c;
                if(i!=7) bitBuffer <<= 1;
            }
            out.write(bitBuffer);
            tree.insertInto(bitBuffer);
        }
        Node node = tree.root;
        while((c = in.read()) != -1) {
            if(c == 1) node = node.right;
            if(c == 0) node = node.left;
            int value = 0;
            if(node.isNYT()) {
                value = readByte(in);
                out.write(value);
                tree.insertInto(value);
                node = tree.root;
            }
            if(node.isLeaf()) {
                value = node.getValue();
                out.write(value);
                tree.insertInto(value);
                node = tree.root;
            }
        }
    }
    catch (IOException e) {
        System.err.println("Error reading bytes");
        e.printStackTrace();
    }
    finally {...}
}

```

Za funkcionisanje Huffman-ovog kodiranja koriste se dve važne klase, a to su Node i Tree.

5.3 Klasa Node

. Polja:

- parent: Referenca na roditeljski čvor u stablu.
- left: Referenca na levog potomka čvora.

- right: Referenca na desnog potomka čvora.
- isNYT: Indikator koji označava da li je čvor NYT (Not Yet Transmitted).
- isLeaf: Indikator koji označava da li je čvor list.
- weight: Težina čvora koja se koristi za Huffmanovo kodiranje.
- index: Indeks ili identifikator čvora.
- value: Vrednost čvora ako je list (u kontekstu Huffmanovog stabla).

Metode klase Node omogućavaju manipulaciju ovim poljima i pružaju osnovne operacije kao što su postavljanje težine, inkrementiranje težine, dobijanje vrednosti i indeksa čvora, kao i provere da li je čvor list ili NYT. Metoda toString služi za generisanje string reprezentacije čvora koja se koristi u debug i ispisu informacija o čvoru.

```
public class Node { 44 usages  ⚡ Miljush *
    public Node parent = null; 14 usages
    public Node left = null; 8 usages
    public Node right = null; 9 usages
    protected boolean isNYT = false; 6 usages
    protected boolean isLeaf = false; 4 usages
    private int weight; 9 usages
    private int index; 8 usages
    private int value; 3 usages
    public Node(Node parent, Node left, Node right, int weight, int index) {...}
    public Node(Node parent) {...}
    public Node(Node parent, int value) {...}
    public boolean isLeaf() {...}
    public boolean isNYT() {...}
    public String toString() {...}
    public void setWeight(int weight) {...}
    public int getWeight() {...}
    public void increment() {...}
    public int getIndex() {...}
    public void setIndex(int index) {...}
    public int getValue() {...}
}
```

5.4 Klasa Tree

Polja:

- root: Koren Huffmanovog stabla.
- NYT: Trenutni NYT čvor (Not Yet Transmitted), koji se koristi za praćenje novih simbola.

- seen: Mapa koja omogućava brz pristup čvoru na osnovu njegove vrednosti (vrednosti simbola).
- order: Lista čvorova organizovana po težini, što pomaže u održavanju redosleda tokom ažuriranja stabla.

Metode:

- Tree(): Konstruktor koji inicijalizuje stablo postavljanjem korena i NYT čvora.
- insertInto(Integer value): Metoda za ubacivanje novog simbola u stablo. Ako simbol već postoji, ažurira odgovarajući čvor; inače, stvara novi čvor.
- contains(Integer value): Proverava da li stablo sadrži dati simbol.
- getCode(Integer c, boolean seen, ArrayList<Boolean> buffer): Generiše Huffmanov kod za dati simbol. Ako simbol nije viđen pre (NYT simbol), vraća dužinu koda; inače, vraća kod pronađenog simbola.
- isEmpty(): Proverava da li je stablo prazno.
- printTree(boolean breadthFirst): Ispisuje stablo, pri čemu se može izabrati da li se koristi širina ili dubina prvo.
- giveBirth(int value): Stvara novi NYT čvor i list čvor za novi simbol, ažurira stablo i vraća prethodni NYT čvor.
- updateTree(Node node): Ažurira stablo nakon ubacivanja ili ažuriranja čvora, prilagođava težine i redosled čvorova.
- maxInWeightClass(Node node): Proverava da li je trenutni čvor najteži u svojoj težinskoj klasi.
- findHighestIndexWeight(Node node): Pronalazi čvor sa najvišim indeksom u istoj težinskoj klasi kao i dati čvor.
- swap(Node newNodePosition, Node oldNodeGettingSwapped): Zamena pozicija dva čvora u stablu.
- updateNodeIndices(): Ažurira indekse čvorova u listi order.
- generateCode(Node in, ArrayList<Boolean> buffer): Generiše Huffmanov kod za dati čvor.
- printTreeDepth(Node node): Rekurzivno ispisuje stablo po dubini.
- printTreeBreadth(Node root): Ispisuje stablo po širini.

```

public class Tree { 7 usages  ⤴ Miljush *

    public Node root; 11 usages
    public Node NYT; // Current NYT node. 11 usages

    // Easily access a node based on its value.
    private Map<Integer, Node> seen = new HashMap<>(); 5 usages
    // Keep nodes in order based on weight.
    private List<Node> order = new ArrayList<>(); 14 usages

    public Tree() {...}
    public void insertInto(Integer value) {...}
    public boolean contains(Integer value) {...}
    public int getCode(Integer c, boolean seen, ArrayList<Boolean> buffer) {...}
    public boolean isEmpty() { return root == NYT; }
    public void printTree(boolean breadthFirst) {...}
    private Node giveBirth(int value) {...}
    private void updateTree(Node node) {...}
    private boolean maxInWeightClass(Node node) {...}
    private Node findHighestIndexWeight(Node node) {...}
    private void swap(Node newNodePosition, Node oldNodeGettingSwapped) {...}
    private void updateNodeIndices() {...}
    private int generateCode(Node in, ArrayList<Boolean> buffer) {...}
    private void printTreeDepth(Node node) {...}
    private void printTreeBreadth(Node root) {...}
}

```

Metoda swap u klasi Tree je odgovorna za zamenu pozicija dva čvora u stablu. Ova zamena obuhvata promenu njihovih pozicija u listi order, a takođe i prilagođavanje njihovih roditeljskih veza u stablu.

Kada se poziva swap(newNodePosition, oldNodeGettingSwapped), prvo se određuju indeksi starih i novih čvorova u listi order. Zatim se prate roditelji oba čvora koji se razmenjuju, kako bi se znalo da li su čvorovi bili levi ili desni roditelji svojih dece. Na osnovu toga, vrši se zamena pozicija u stablu:

1. Ako je newNodePosition bilo desno dete svog roditelja, on sada postaje levo dete oldNodeGettingSwapped-a, i obrnuto.
2. Slično, ako je oldNodeGettingSwapped bilo desno dete svog roditelja, sada postaje levo dete newNodePosition-a.

Nakon fizičke zamene pozicija u stablu, ažuriraju se roditeljske veze oba čvora: oldNodeGettingSwapped dobija novog roditelja (newParent), dok newNodePosition dobija

starog roditelja (oldParent). Takođe, ažuriraju se indeksi čvorova u listi order, kako bi odražavali njihove nove pozicije.

```
private void swap(Node newNodePosition, Node oldNodeGettingSwapped) { 1 usage  ⤴ Miljush *
    int newIndex = newNodePosition.getIndex();
    int oldIndex = oldNodeGettingSwapped.getIndex();

    Node oldParent = oldNodeGettingSwapped.parent;
    Node newParent = newNodePosition.parent;

    boolean oldNodeWasOnRight, newNodePositionOnRight;
    oldNodeWasOnRight = newNodePositionOnRight = false;

    if(newNodePosition.parent.right == newNodePosition) {
        newNodePositionOnRight = true;
    }
    if(oldNodeGettingSwapped.parent.right == oldNodeGettingSwapped) {
        oldNodeWasOnRight = true;
    }
    if(newNodePositionOnRight) {
        newParent.right = oldNodeGettingSwapped;
    }
    else{
        newParent.left = oldNodeGettingSwapped;
    }
    if(oldNodeWasOnRight) {
        oldParent.right = newNodePosition;
    }
    else {
        oldParent.left = newNodePosition;
    }
    oldNodeGettingSwapped.parent = newParent;
    newNodePosition.parent = oldParent;
    order.set(newIndex, oldNodeGettingSwapped);
    order.set(oldIndex, newNodePosition);
    updateNodeIndices();
}
```

6. Tabela ulaza i izlaza

Tip fajla	Veličina pre kompreisje(B)	Veličina nakon kompresije(B)	Odnos kompresije
.txt	55556	6946	7.9982724
.jpg	3802207	3775438	1.0070903
.pdf	35269	34209	1.030986

7. Zaključak

Kompresija fajlova je ključni proces koji omogućava efikasno skladištenje i prenos podataka tako što smanjuje veličinu originalnog fajla. Adaptivno Huffmanovo kodiranje, kao što je Vitterova metoda, predstavlja naprednu tehniku koja se prilagođava dinamičnim promenama u distribuciji simbola tokom kodiranja. Ova tehnika je posebno korisna kada je statistika pojavljivanja simbola u podacima nepredvidiva ili se menja tokom vremena.

Zahvaljujući adaptivnosti, Huffmanovo stablo se kontinuirano prilagođava novim informacijama o simbolima koje kodira. To omogućava da se češće korišćeni simboli kodiraju kraćim kodovima, dok se retki simboli kodiraju dužim kodovima, što dovodi do efikasnijeg ukupnog kompresijskog rezultata. Ova fleksibilnost čini adaptivno Huffmanovo kodiranje pogodnim za širok spektar aplikacija gde je potrebna visoka efikasnost pri kompresiji podataka bez unapred poznate statistike.

8. Literatura

1. Data Compression (3rd Edition) - The Complete Reference
2. https://www.stringology.org/DataCompression/ahv/index_en.html
3. <https://www.javatpoint.com/java-swing>