****

**PROJEKT**

**Pozyskiwanie i odzyskiwanie danych elektronicznych**

**Kodowanie Huffmana**

**Jakub Kuśnierz**

**w61674**

**Informatyka, II semestr studiów magisterskich**

1. **Cel projektu**

Celem projektu było zaprojektowanie, wykonanie i przetestowanie aplikacji realizującej metodę bezstratnej kompresji Huffmana. Środowiskiem które zostało wybrane do wykonania całości został język programowania Java w wersji JDK 1.8. Wykorzystane zostały tylko funkcjonalności wbudowane w kod źródłowy języka, nie potrzebne były żadne biblioteki zewnętrzne. Użyto między innymi implementacji struktur danych jak mapa oraz kolejka priorytetowa oraz interfejsu Comparable służącego do porównywania obiektów. Programem w którym powstawał kod projektu był Intellij Idea produkowany przez firmę Jetbrains i wykorzystany w pełnej wersji na podstawie licencji udostępnianej dla studentów.

1. **Kodowanie Huffmana**

Kodowanie Huffmana jest jedną z metod kompresji bezstratnej. Zostało opracowane w 1952 roku przez Amerykanina Davida Huffmana. Algorytm nie jest doskonały, jednak używany ze względu na prostą w porównaniu do innych algorytmów implementacje oraz brak ograniczeń patentowych. Kodowanie Huffmana jest przykładem wykorzystania algorytmu zachłannego, którego działanie polega na wybraniu w każdym kroku najlepiej rokującego w danym momencie rozwiązania częściowego.

Kompresja Huffmana polega na zastępowaniu znaków kompresowanego pliku krótszymi kodami, a dokładniej ciągami bitów. Dzięki temu wynikowy ciąg bajtów jest znacznie krótszy niż ciąg wejściowy. Znaki są kodowane na 8 bitach, można na nich zapisać do 256 znaków. Każdy standardowy znak na przypisany swój numer w tablicy kodów ASCII. Rzadko kiedy jednak wszystkie z nich występują w pliku. Najczęściej są tam tylko litery, cyfry i kilka znaków specjalnych. Dzięki temu można wykorzystać mniej bitów i zaoszczędzić przez to dużo miejsca.

Kodowanie Huffmana polega na przypisaniu znakom liczb porządkowych, np. załóżmy, że nasz plik został zapisany za pomocą znaków alfabetu A={a, b, c, d, e). Alfabet ten składa się z 5 znaków jest to dużo mniej niż 256, nie potrzebujemy więc aż 8 bitów, wystarczą nam 3 bity, gdyż 23=8, za pomocą 3 bitów można zapisać 8 znaków. Załóżmy, że nasz plik ma dokładnie 1MB, czyli 1 048 576 bajtów. Każdy bajt jest kodowany na 8 bitach, czyli nieskompresowany plik zajmuje 8 \* 1 048 576 = 8 388 608 bitów. Jednak nam wystarczą po 3 bity na znak. Przypisujemy więc każdemu znakowi liczbę porządkową (zapisaną binarnie): a=000, b=001, c=010, d=011, e=100. Zamiast używać 8 bitowych kodów ASCII używamy własnych 3 bitowych kodów. Odnieśliśmy na tym duże korzyści: 1 048 576 \* 3 = 3 145 728 bitów, czyli 37,5% pliku wejściowego. Musimy zauważyć jednak, że tak dobry wynik osiągamy tylko dlatego, że nasz alfabet miał jedynie 5 znaków. Im większy alfabet, tym gorszy wynik. Przy alfabecie składającym się ze wszystkich 256 znaków będziemy potrzebowali 8 bitów, czyli tyle ile normalnie.

Główną wadą zastosowania algorytmu Huffmana jest konieczność przesłania całego drzewa aby możliwe było odtworzenie kodów jakie przypisaliśmy poszczególnym znakom. Bez niego dekompresja pliku staję się niemożliwa.

Istnieją równe modyfikację kodowania Huffmana m.in. algorytm dynamicznego kodowania Huffmana lub kodowanie po 2 znaki na raz. W projekcie została pokazana standardowa wersja algorytmu, oparta na statycznym kodowaniu.

Własności kodu Huffmana są następujące:

- jest kodem prefiksowym; oznacza to, że żadne słowo kodowe nie jest początkiem innego słowa

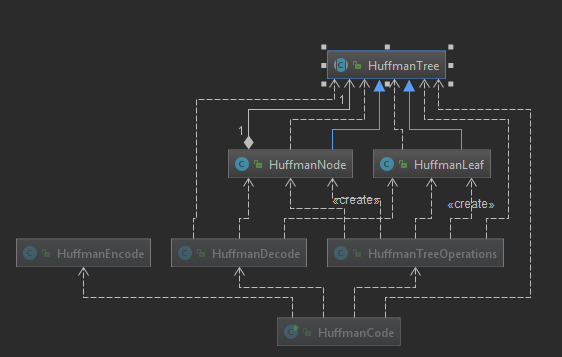
- średnia długość słowa kodowego jest najmniejsza spośród kodów prefiksowych

- słowa kodu dwóch najmniej prawdopodobnych symboli mają równą długość.

1. **Diagram klas aplikacji**

Aplikacja składa się z dwóch głównych modułów. Jest to model do których należy klasa abstrakcyjna *HuffmanTree* implementująca interfejs *Comparable* oraz klasy reprezentujące liście i węzły drzewa dziedziczące po *HuffmanTree* czyli *HuffmanNode* i *HuffmanLeaf.* Zadaniem tych klas jest odwzorowanie drzewa wraz z możliwością przechowywania w nim potrzebnych informacji. Drugim głównym modułem są operację które możemy wykonywać na wprowadzonym tekście. Należy do nich zbudowanie drzewa, stworzenie mapy z przypisanymi kodami, wypisanie drzewa do celów poglądowych, zakodowanie oraz zdekodowanie tekstu. Całość funkcjonalności użyta została w głównej klasie aplikacji o nazwie HuffmanCode. Została w niej również stworzona metoda zamiany zakodowanego tekstu na nowe znaki z tabeli kodów ASCII ukazujący lepsze ukazanie naturalnemu odbiorcy zaoszczędzenie miejsca poprzez konwersje niż ciąg bitów.

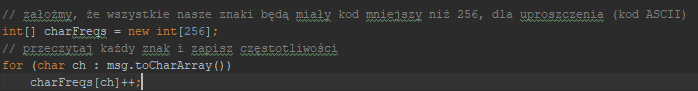
Diagram klas wraz z zachodzącymi relacjami wygenerowany za pomocą wykorzystywanego programu Intellij Idea został pokazany na zdjęciu poniżej.



1. **Opis wykonywanych kroków algorytmu**

Pierwszym krokiem działania algorytmu jest wczytanie tekstu, który chcemy poddać konwersji algorytmem Huffmana. Następnie przechodzimy w pętli po każdym znaku wczytanego tekstu zwiększając wystąpienie każdego znaku o 1 w momencie jego napotkania w tekście. Dla uproszczenia zakładamy, że wszystkie nasze znaki będą miały kod mniejszy niż 256 czyli będą należeć do tablicy kodów ASCII. Spowodowane to jest koniecznością sprecyzowania kodowania na potrzeby projektu. Początkowo standardy zawierały 127 znaków, które były wystarczające, obecnie najnowszy standard UNICODE 12 zawiera ich już ponad 110 tysięcy. Zawarte w tym są litery z różnych języków w tym chiński oraz emotikony. Znaki te powstają z połączenia innych znaków zapisywanych na 1 bajcie przez co mają większe rozmiary. Na zapisanie największego pojedynczego znaku a zasadzie emotikony potrzeba 11 bajtów. My jednak skupimy się na znakach zapisywanych na jednym bajcie, dlatego użyjemy tablicy kodów ASCII.

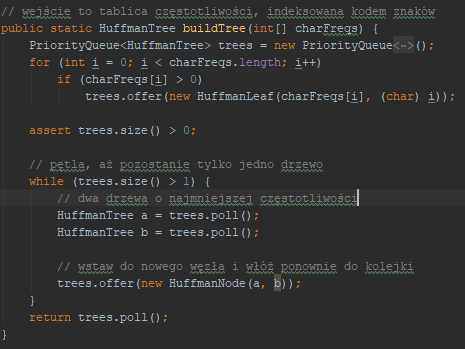
Poniżej przedstawiona została operacja przypisania częstotliwości do znaków według ich występowania w tekście.



Kolejnym krokiem gdy mamy już tablicę z częstotliwością występowania znaków jest zbudowanie tak zwanego drzewa Huffmana. Odbywa się to za pomocą specjalnie przygotowanej metody, która jako parametr wejściowy przyjmuję wcześniej stworzoną tablice.

W metodzie pierwsze zostaje zainicjalizowana kolejka priorytetowa, którą wykorzystamy do wykonania drzewa. Następnie są do niej dodawane liście, którymi są wszystkie znaki występujące co najmniej raz w początkowym tekście. Są one do kolejki według priorytetu. Czym więcej razy dany znak wystąpił tym wyżej będzie w kolejce. W tym momencie nasze drzewo składa się z samych liści, które są na jednym poziomie. Kolejnym krokiem, który będziemy wykonywać aż do pozostania nam w kolejce jednego węzła które będzie korzeniem głównym jest iteracja po drzewie. Podczas niej wybieramy dwa drzewa o najmniejszej częstotliwości występowania i łączymy je w jedno sumując częstotliwości, a następnie wstawiamy nowe drzewo do naszej kolejki w odpowiednie miejsce według nowej częstotliwości. Operacje tą wykonujemy aż do uzyskania korzenia naszego drzewa, które jest obiektem wynikowym zwracanym przez funkcję.

Kod metody służącej budowaniu drzewa został pokazany poniżej.

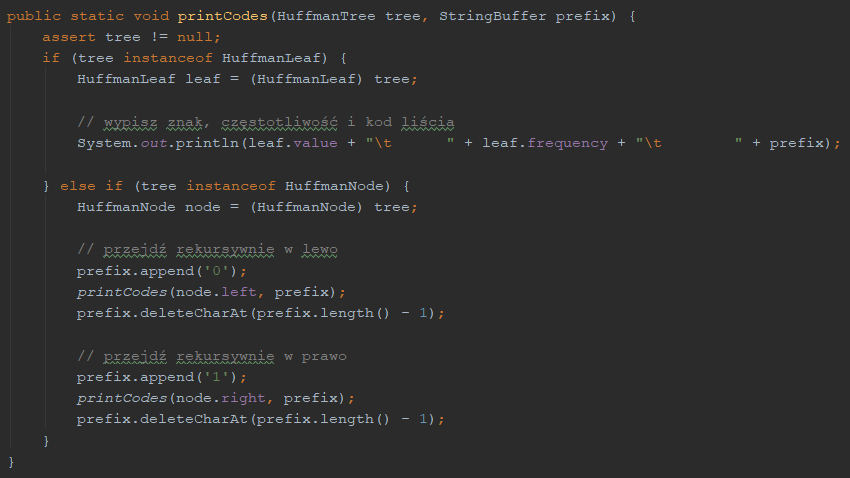


Gdy mamy już zbudowane drzewo z nadanymi nowymi kodami, możemy przejść do następnego kroku w którym wypiszemy w konsoli całe drzewo. Zostanie ono wypisane w postaci listy zawierającej dany znak, ilość jego wystąpienia w tekście oraz kod który został mu przypisany w drzewie binarnym. Wypisywanie odbywa się za pomocą metody umieszonej na obrazku poniżej. Przyjmuję ona na wejściu dwa parametry, którymi są węzeł drzewa oraz prefiks (kod) znaku. Jest to metoda rekurencyjna czyli wywołująca wewnątrz samą siebie więc parametry za każdym razem przyjmą różną wartość. Pierwsze wywołanie następuję z głównym korzeniem drzewa oraz pustym ciągiem jako prefiks. Wewnątrz metody sprawdzamy czy węzeł ma potomków (jeżeli ma jest instancją klasy węzła, jeżeli nie to liścia). Gdy nie ma to wypisujemy wartość według wcześniej podanego schematu. Gdy ma potomków to przechodzimy dołączamy odpowiedni prefiks do obecnego i wywołujemy rekursywnie metodę zaczynając od strony lewej. Drzewo wypiszę się od najniższych liści z lewej strony przez całą lewą stronę w górę, a następnie prawą stroną od „Góry do dołu”. Przykład wypisania zostanie pokazany w sekcji pokazującej przykłady działania aplikacji w dokumentacji.

Metoda ta jest bardzo podobna do metody zapisującej pary znak - nowy kod do mapy (tablicy haszującej), która posłuży nam do zakodowania wpisanego tekstu bez potrzeby przeszukiwania drzewa binarnego. Metoda wypisują mogła by być połączona z uzupełniającą mapę nowymi kodami jednak wtedy nie zachowamy zasady pojedynczej odpowiedzialności metody i będziemy zmuszenie wypisywać znaki w konsoli przy każdym uzupełnianiu mapy, czego wolelibyśmy uniknąć, zostawiając wypisywanie tylko do celów poglądowych ukazujących działanie algorytmu.

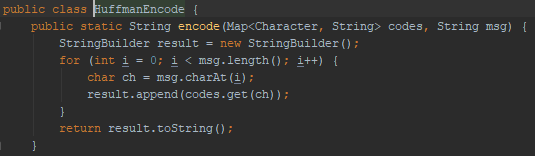
Poniżej znajdują się obrazki z kodem odpowiadający za pierwsze wywołanie opisanych metod oraz implementacją metody wypisującej znaki na konsoli.





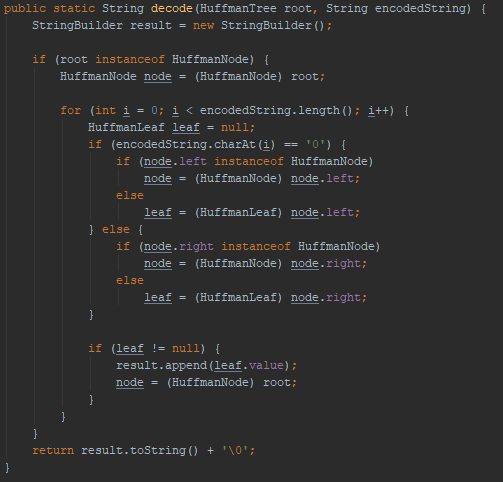
Gdy mamy już wszystko przygotowane możemy przystąpić do skompresowania wpisanego tekstu. Odbywa się to za pomocą metody, która przyjmuję na wejściu mapę z kodami przypisanymi do znaków oraz wiadomość, którą należy poddać kompresji, a zwraca wiadomość powstałą w wyniku przeprowadzonej operacji. Wewnątrz niej tworzymy obiekt StringBuildera, do którego dołączać będziemy kolejne kody znaków. Następnie przechodzimy w pętli przez wszystkie znaki wiadomości wejściowej i dla każdego znaku wyszukujemy w mapie nowy kod dołączając go do zainicjalizowanego wcześniej obiektu. Po ukończeniu pętli zwracamy powstały kod w formie tekstu.

Kod metody kompresującej wiadomość został pokazany na poniższym obrazku.



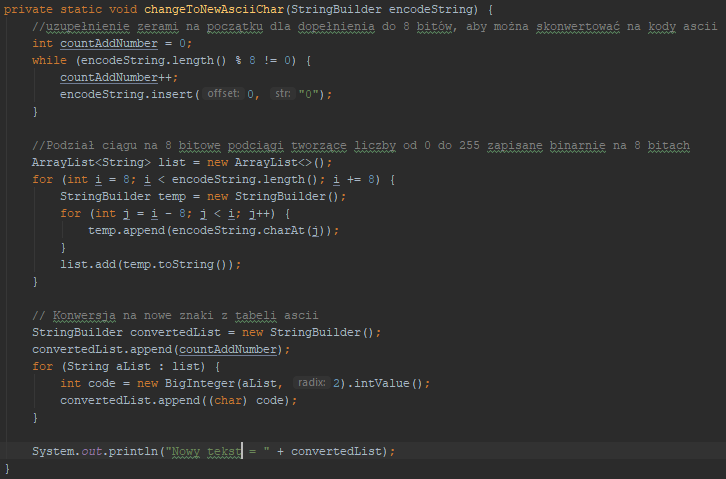
Kolejnym krokiem będzie odwrócenie kompresji, czyli powrót do podstawowej wiadomości z zakodowanego ciągu zwracanego przez poprzednio opisywaną metodę. W tym celu skorzystamy z kolejnej funkcji i powrócimy do wykorzystania drzewa Huffmana. Taką operację można wykonać również przy użyciu mapy, tak samo jak do operacji kodowania można użyć drzewa, jednak dla celów projektu skorzystałem na przemian z obu metod, aby wykazać ich działanie i przykładową implementację. Funkcja dekodująca przyjmuję jako parametry wejściowe główny węzeł zbudowanego drzewa oraz skompresowaną wiadomość. Wewnątrz niej najpierw inicjalizujemy obiekt StringBuildera do którego będziemy dopisywać kolejno zdekompresowane znaki. Później iterujemy w pętli po wiadomości otrzymanej na wyjściu sprawdzając każdy znak w sposób następujący: sprawdzamy czy on jest liściem jeżeli tak to wypisujemy symbol wejściowy z liścia, gdy jest węzłem to w zależności od znaku schodzimy w lewo gdy było to 0 lub w prawo gdy znakiem było 1. Przechodzimy do następnego znaku zaczynając od samej góry drzewa jeżeli wcześniej mieliśmy liść lub od węzła do którego zeszliśmy. Operację powtarzamy aż do przejścia przez całą wiadomość. Po zakończeniu zwracamy wartość wynikową metody, którą jest zdekompresowana wiadomość.

Funkcję do dekompresji pokazano na obrazku poniżej.



Dodatkowym algorytmem, który zaimplementowałem jest zamiana ciągu binarnego powstałego w wyniku kompresji na nowe znaki z tablicy UNICODE (zakres od 0 do 255). Tutaj możemy zauważyć ile taki ciąg będzie krótszy od wejściowej wiadomości. Aby zrozumieć taki zabieg musimy wiedzieć jak działa komputer. Wysyła on dwa stanu określane jako 0 albo 1 (napięcie lub brak napięcia). Nasz zakodowany kod składał się z 0 i 1 więc możemy wysłać wiadomość jako strumień bajtów zyskując na ilości potrzebnych do wysłania danych. Jeżeli jednak nasz ciąg zapisalibyśmy jako plik tekstowy jego rozmiar zrobi się większy niż pliku przed kompresją. Spowodowane jest to tym, że 0 i 1 zostaną potraktowane wtedy jako symbole i zapisane jako kody z tablicy ASCII, gdzie każdy będzie zajmował 8 bitów. Aby tego uniknąć przypiszemy każdym 8 znakom naszego ciągu nowy znak, który odpowiadał będzie wartości binarnej danego ciągu. Musimy wprowadzić jednak pewien zabieg normalizujący. Nasz ciąg nie zawszę będzie dzielił się na 8 znakowe części bez reszty. Dlatego dodamy na początku ciągu tyle zer aby uzupełnić go do prawidłowej postaci. Dla poprawnego odkodowania później musimy dodać na początku symbol z tablicy kodów ASCII informujący ile zer dodaliśmy aby wykonać odpowiednie przesunięcie bitowe podczas dekodowania. Metoda dekodująca będzie musiała odczytać pierwszy symbol którym jest cyfra, następnie przesunąć się w prawo o taką wartość jaką ona ma i od tego miejsca rozpocząć dekodowanie.

Taki zabieg kodowania jest niestandardowy i ważnym jest wprowadzenie takich samych zasad uzupełniania znaków zarówno w metodzie służącej do kodowania jak i odkodowywania. Inaczej otrzymamy niepoprawną wiadomość, którą poddaliśmy kompresji. Przykład kodowania zostanie pokazany w następnym rozdziale, natomiast poniżej ukazany został kod metody służącej do zakodowania za pomocą nowych znaków. Metoda dekodująca była by odwrotnością tej operacji, dlatego nie było potrzeby jej implementacji z uwagi na niski poziom trudności.

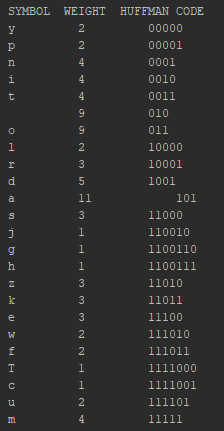


1. **Przykład działania**

Do testów wykorzystamy krótką wiadomość tekstową, która brzmi następująco:

*To jest przyklad wiadomosci ktora zostanie poddana zakodowaniu algorytmem huffmana*

Ważnym jest aby nie zawierała ona polskich znaków ponieważ nie należą one do tablicy ASCII. Następnie zbudujemy i wypiszemy na jej podstawie drzewo Huffmana, które wygląda następująco:



Widzimy z niego, że najczęściej występującymi znakami była litera a (11 razy), o (9 razy) oraz spacja która również wystąpiła 9 razy. Otrzymały one najkrótsze kody zawierające zaledwie po 3 znaki.

Kolejnym krokiem będzie poddanie zabiegowi zakodowania po którym otrzymamy poniższy ciąg bitów:

*11110000110101100101110011000001101000001100011101000000110111000010110010101110100010101100101111111011110001111001001001011011001101110001101010110100111100000111010001001011100010000010111001100110100011010101101010111011011100101111101010100010010111101010101100001100110011100010000000111111111100111110101100111111101111011111011111111010001101*

Dla porównania poniżej zapisana binarnie została oryginalna wiadomość.

1010100011011110010000001101010011001010111001101110100001000000111000001110010011110100111100101101011011011000110000101100100001000000111011101101001011000010110010001101111011011010110111101110011011000110110100100100000011010110111010001101111011100100110000100100000011110100110111101110011011101000110000101101110011010010110010100100000011100000110111101100100011001000110000101101110011000010010000001111010011000010110101101101111011001000110111101110111011000010110111001101001011101010010000001100001011011000110011101101111011100100111100101110100011011010110010101101101001000000110100001110101011001100110011001101101011000010110111001100001

Różnica między poniższymi ciągami wynosi dokładnie 305 znaków. I daję nam zysk przy takim doborze aż o 47 % w porównaniu do wiadomości wejściowej.



Na końcu dodatkowy mechanizm zmieniający nasz zakodowany ciąg w nowy ciąg znaków. Jak widzimy jest on dużo krótszy niż nasz ciąg wejściowy. Mimo konieczności dodania dodatkowych znaków.



1. **Podsumowanie**

Implementacja kodowania Huffmana jest bardzo fajnym przykładem, który możemy wykorzystać w aplikacjach do zmniejszenia rozmiarów naszego tekstu. Jego wydajność zależy od znaków z których składa się widomość poddawana kompresji. Im mniejsze zróżnicowanie tym większe efekty uzyskamy. Duża wadą tego rozwiązania jest konieczność przesyłania drzewa Huffmana do każdego zdekompresowania pliku. Jednak można w aplikacjach próbować to rozwiązać. Gdy mamy dużo podobnych plików w systemie (zawierają np. same cyfry lub litery od A do G) możemy zdecydować się na zbudowanie wspólnego drzewa. Zmniejszymy wtedy wydajność kompresji, ale zaoszczędzimy czas na każdorazowe budowanie drzewa oraz konieczność jego wielokrotnego przesyłania. Wszystko zależy od tego co chcemy osiągnąć i jakie wyniki nas zadowolą. Czasami nie potrzebujemy maksymalnej kompresji tylko ważniejsza dla nas jest szybkość działania przy zadowalającym poziomie zmniejszenia plik. Po zbudowaniu drzewa możemy trzymać kody w drzewie które będziemy przeglądać lub w tablicy haszującej. Obie te struktury nadają się do tego ze względu na niską złożoność operacji wyszukiwania.

1. **Literatura**
2. <https://pl.wikipedia.org/wiki/Kodowanie_Huffmana>
3. <https://eduinf.waw.pl/inf/alg/001_search/0121a.php>
4. <http://www.algorytm.org/algorytmy-kompresji/kody-huffmana.html>
5. <https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>