Struktury danych

- widzieliśmy jedną strukturę danych w pamięci: listę
- najczęściej są jedno-, dwukierunkowe lub cykliczne
- widzieliśmy implementację prostej bazy danych jako listy dwukierunkowej, wiele obiektów można efektywnie implementować jako listy
- korzystając z listy nie musimy za każdym razem od początku definiować odpowiedniej klasy
- listy dostępne są jako gotowe *szablony* - zajmiemy się tym tematem wkrótce

Struktury danych

- inną strukturą danych jest drzewo. składa się z węzłów, w których przechowywane są dane, oraz połączeń między węzłami. Każdy węzeł ma jednego *przodka* i pewną, być może zerową liczbę *potomków*. Jest dokładnie jeden węzeł, który nie ma przodka, to jest tak zwany *korzeń*. Terminologia nie jest dokładnie ściśle ustalona, ale widać o co chodzi
- węzeł, który ne ma potomków nazywa się *liściem*, a taki, który ma potomków nazywa się węzłem wewnętrznym
- często spotykanym rodzajem drzewa jest drzewo binarne, czyli takie, którego każdy węzeł ma co najwyżej 2 potomków

Struktury danych

- implementacja drzewa binarnego może być następująca. Węzły są obiektami następującej klasy

```
class my_node
{
    my_node *up, *left, *right;
          ...
    inne dane/metody
          ...
};
```

- tak naprawdę wskaźnik na przodka nie jest potrzebny. Tworzenie, usuwanie, przeszukiwanie drzewa w zasadzie zawsze zachodzi od korzenia w dół
- korzeń poznajemy po tym, że jego wskaźnik `up` jest `NULL`, podobnie liście poznajemy po tym, że oba wskaźniki `left` i `right` są `NULL`
- program musi pamiętać wskaźnik do korzenia. Do wszystkich innych węzłów można dojść począwszy od korzenia

Struktury danych

- przeglądanie drzewa może przebiegać zgodnie z algorytmem DFS (Depth First Search). Algorytm ten ma kilka wariantów, rozważmy jeden. Piszemy procedurę, która jest wywoływana rekurencyjnie

```
void czytaj(my_node *p)
{
    if ( p )    //jeśli wskaźnik nie jest NULL
    {
        odczytaj dane
        czytaj( p -> left );
        czytaj( p -> right );
    }
}
```

- aby przejść całe drzewo wywołujemy procedurę `czytaj()` ze wskaźnikiem na korzeń
- łatwo zauważyć, czemu algorytm nazywa się Depth First

Struktury danych

- inna możliwość to algorytm BFS (Breadth First Search). Nie będziemy go implementowali, ale w tym algorytmie drzewo przeglądamy *poziomami*. Odwiedzamy węzeł, i wszystkich jego potomków zapamiętujemy. Można to zrobić przy pomocy struktury zwanej *kolejką*. Jest to lista, w której elementy dodajemy zawsze na końcu, a usuwamy zawsze z początku (młodzież w dzisiejszych czasach może nie „jarzyć" pojęcia kolejki :-))
- odwiedzamy więc węzeł, i wszystkich potomków dodajemy do kolejki. Następnie przechodzimy do kolejnego węzła w kolejce. Oczywiści, jeżeli odwiedziliśmy liść, to nic już do kolejki nie dochodzi
- łatwo zauważyć, czemu algorytm nazywa się Breadth First

Kompresja Huffmana

- teksty, czyli ciągi znaków będziemy chcieli zapisać przy pomocy minimalnej ilości bitów
- poszczególnym literom przypisujemy ciągi bitów, ale różnej długości. Literom, których jest dużo przypiszemy krótkie ciągi, a literom występującym w tekście rzadko dłuższe ciągi
- mając gotowy kod, czyli poszczególnym znakom alfabetu przypisany konkretny ciąg binarny, możemy go zaimplementować jako drzewo binarne. Węzły wewnętrzne drzewa nie mają przypisanych znaków, natomiast każdy liść ma przypisany znak występujący w alfabecie
- począwszy od korzenia, przechodząc do lewego potomka do ciągu binarnego dopisujemy 0, natomiast przechodząc do prawego potomka dopisujemy 1. W ten sposób każda ścieżka od korzenia do liścia reprezentuje pewien unikalny ciąg binarny
- procedura kodowania (czyli zamiany ciągu znaków alfabetu na ciąg bitów) i odkodowywania (czyli zamiany ciągu bitów na ciąg znaków alfabetu) sprowadza się do przeszukiwania drzewa.
- Przykład: prog1.cpp

Kodowanie Huffmana

- kodowanie Huffmana to kodowanie ze zmienną długością kodu, bezprefiksowe, w którym kod znaku jest tym dłuższy, im rzadziej znak występuje w tekście. Kodowanie to stosowane jest do ciągu bajtów (niekoniecznie tekstów), i ma efekt bezstratnej kompresji. Jest częstym końcowym etapem innych operacji (na przykład zapisu obrazu w formacie `.jpeg`. Kiedy otrzymywana jest ostateczna reprezentacja kompresowanego obrazu, powstały ciąg bajtów kodowany jest kodem Huffmana, aby dodatkowo ścisnąć plik.
- do stworzenia tabeli kodów potrzebna jest tabela ilości poszczególnych znaków w kodowanym tekście. Mając taką tabelę, tworzymy węzeł (na razie liść) odpowiadający każdemu znakowi, a węzły ustawiamy w listę, posortowaną według ilości wystąpień znaków

Kodowanie Huffmana

- następnie, rekurencyjnie, pobieramy dwa węzły z początku listy, i łączymy je w nowy węzeł. Dwa stare węzły zostają dwoma potomkami nowego węzła, ilości wystąpień potomków sumują się do łącznej ilości wystąpień nowego węzła, a nowy węzeł jest umieszczany na właściwym miejscu listy
- iteracje kończą się, kiedy na liście zostanie tylko jeden węzeł, będzie on korzeniem drzewa
- nietrudno przekonać się, że tak powstałe drzewo realizuje ideę kodowania Huffmana
- Przykład: `prog2.cpp`