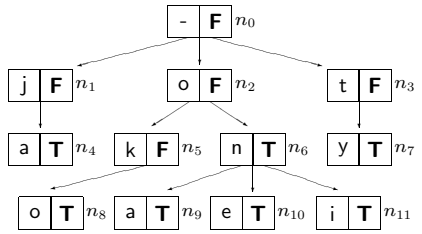
Drzewa wielokierunkowe *trie*

Do zastosowań, w których kluczem są napisy znakowe (stringi), takich jak na przykład budowa słowników, przydatne są specjalnego rodzaju drzewa, zwane *tries*. (Nie ma przyjętego polskiego odpowiednika tej nazwy). *Tries* są drzewami uporządkowanymi i wielokierunkowymi, czyli o rzędzie ≥ 2. W węzłach drzewa przechowywane są pojedyncze znaki, a napisy „odczytuje się" w ścieżkach drzewa biegnących od korzenia. Dla słów (stringów) zbudowanych z liter łacińskiego alfabetu, drzewa *trie* będą miały rząd 26. Dla alfabetu polskiego rząd wyniesie 35.

Przykładowe *trie* dla słów: „ja", „ty", „oko", „on", „ona", „one", „oni"



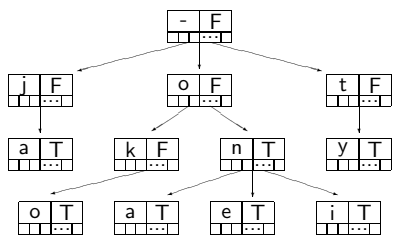
W oczywisty sposób, czas dostępu do każdego słowa przechowywanego na takim drzewie, mierzony liczbą porównań kolejnych liter, jest proporcjonalny do, i ograniczony długością, najdłuższego przechowywanego słowa. Ponieważ dla większości zastosowań istnieje maksymalna długość stringa, niezależnie od ich liczby, z punktu widzenia miary złożoności obliczeniowej jest to procedura o stałym czasie wykonania, czyli $O(1)$. Widzimy więc, że *trie* są bardzo sprawnymi strukturami do szybkiego sprawdzania zbiorów słów (ich występowania, i własności).

Algorytmy tworzenia i modyfikacji drzew *trie* są trywialne i polegają, oprócz dodawania w razie potrzeby węzłów do drzewa, na ustawianiu i kasowaniu flag logicznych decydujących o tym, czy w danym węźle jest zakończenie poprawnego stringa czy nie. Jak się okazuje, pewne znaczenie ma za to zastosowany typ danych.

Drzewa *trie* — reprezentacja

Deklaracja struktury danych dla takich drzew mogłaby mieć postać wykorzystującą podobny jak dla B-drzew mechanizm tablicy wskaźników do poddrzew:

```
TYPE
   Trie = ^TrieElem;
   TrieElem =
      RECORD
         Znak: T_Info;
         Slowo: BOOLEAN;
         Potomki:
            ARRAY['a'..'z']
               OF Trie;
      END;
```



Drzewa *trie* — wymagania pamięciowe

Reprezentacja drzewa z wykorzystaniem tablicy wskaźników jest jednak wyjątkowo niekorzystna pamięciowo. Łatwo obliczyć, że w dowolnym *trie* rzędu k będzie k razy więcej wskaźników niewykorzystanych, niż wykorzystanych, ponieważ wykorzystanych wskaźników jest tyle ile węzłów drzewa (minus korzeń), natomiast każdy węzeł ma k wskaźników.

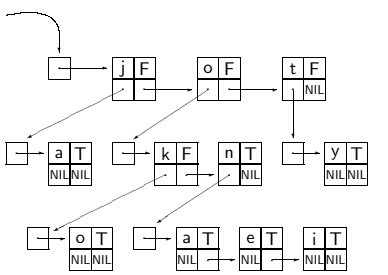
Na przykład, chcąc zbudować słownik z 200 tysiącami słów, maksymalnie 16-znakowych, *trie* rzędu 35 mogłoby zajmować, przy założeniu, że co drugi węzeł reprezentuje słowo (a co drugi tylko strukturę drzewa), około $400,000 \times 35 = 14$ milionów wskaźników czyli dla 32-bitowego komputera ponad 50 megabajtów. To nie jest wielkość astronomiczna, ale pamiętajmy, że 97% tego będą zajmowały same puste wskaźniki. Jednocześnie 200 tysięcy słów można zapisać na około 2 megabajtach pamięci, zakładając średnią długość słowa około 9 znaków.

Trie są zatem bardzo nieoszczędne pod względem zużycia pamięci.
A raczej, nieoszczędna jest tablicowa reprezentacja struktury drzewa.

Drzewa *trie* — reprezentacja (cd.)

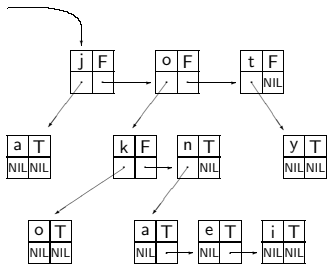
Aby zaoszczędzić pamięć należy zastąpić statycznie alokowane tablice strukturami dynamicznymi, czyli listami. W każdym węźle drzewa możemy umieścić listę wskaźników do jego poddrzew. Jednak zawartość informacyjna i wskaźniki do elementów niższego poziomu muszą być związane ze wskaźnikami do poddrzew, a nie ze wskaźnikiem do listy elementów danego węzła. Prowadzi to do drzewa złożonego z dwóch rodzajów węzłów.

```
TYPE
   Trie = ^TrieList;
   TrieList = ^TrieElem;
   TrieElem =
      RECORD
         Znak: T_Info;
         Slowo: BOOLEAN;
         Potomki: Trie;
         Sasiady: TrieList;
      END;
```



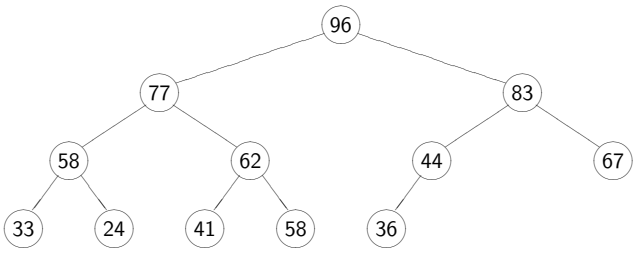
Widać jednak, że powyższa struktura jest dziwna, ponieważ zarówno zawartość informacyjna jak i większość struktury *trie* jest ulokowana w elementach. Dodatkowe wskaźniki są zbędne i można się ich pozbyć, otrzymując prostszą i bardziej naturalną reprezentację, zwaną lewy-skrajny-potomek/prawy-sąsiad. W tej reprezentacji zbędny jest również dotychczas pusty korzeń *trie*:

```
TYPE
   Trie = ^TrieElem;
   TrieElem =
      RECORD
         Znak: T_Info;
         Slowo: BOOLEAN;
         Potomek, Sasiad: Trie;
      END;
```



Stogi — definicja

Stogiem nazywamy kompletne drzewo binarne, w którym każdy węzeł zawiera element o wartości nie mniejszej od wartości w obydwu jego poddrzewach. Drzewo binarne jest **kompletne**, jeśli zawiera na wszystkich poziomach pełne ilości węzłów, z wyjątkiem najniższego poziomu, na którym elementy są ułożone od lewej strony bez żadnych luk.

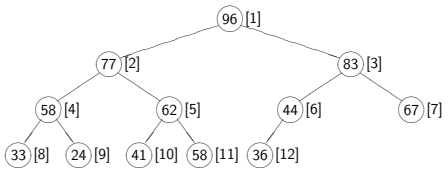


Tablicowa reprezentacja stogu

$2 \times N$ = indeks lewego poddrzewa elementu N

$2 \times N + 1$ = indeks prawego poddrzewa elementu N

$\lfloor \frac{N}{2} \rfloor$ = indeks rodzica elementu N



indeks	1	2	3	4	5	6	7	8	9	10	11	12
zawartość	96	77	83	58	62	44	67	33	24	41	58	36

Tworzenie stogów

Dodawanie elementu:

1. wmontuj element na najniższym poziomie stogu (lub poniżej, jeżeli najniższy poziom jest pełny), na pierwszej wolnej pozycji od lewej,
2. jeśli wmontowany element jest większy od swojego poprzednika (rodzica), to zamień go z poprzednikiem i powtórz ten krok.

Usuwanie elementu:

1. znajdź na stogu element do usunięcia, co może wymagać pełnego przeglądu stogu, ponieważ stóg nie ma uporządkowania pozwalającego dowolny element łatwo odszukać,
2. zastąp element do usunięcia skrajnie prawym elementem ostatniego (najniższego) poziomu stogu, ten element jest w pewnym sensie „ostatnim" elementem stogu, choć nie musi być najmniejszym,
3. sprawdź, czy element „ostatni" wstawiony w miejsce elementu usuwanego nie jest większy od swojego poprzednika (rodzica), albo mniejszy od któregoś ze swoich poddrzew (potomków); jeśli tak, to zamień go z tym rodzicem lub potomkiem i powtórz ten krok.

Szybki test

Czy następująca tablica jest stogiem:

100 99 98 97 96 95 94 93 92 91 90

a ta:

100 100 90 100 100 65 75 100 100 100 100 25 35 45 55

a ta:

10 20 25 50 60 30 40 75 85 35

a ta:

100 90 □ 80 □ □ □ 70

Dla tych z powyższych tablic, które nie są stogami, czy potrafisz pokazać operacje przekształcające je na stogi?

Operacje na stogach

Dodawanie i usuwanie elementów są dla stogów operacjami $O(\log N)$, podobnie jak dla drzew zrównoważonych. Istnieje ponadto procedura pozwalająca dowolne kompletne drzewo binarne przekształcić na stóg za pomocą $O(N)$ operacji propagacji. Zatem budując stóg z N elementów opłaca się umieścić elementy na stogu w dowolnym porządku, a następnie zamienić tak powstałe drzewo na stóg. W ten sposób otrzymamy N-elementowy stóg w $O(N)$ krokach, ponieważ $O(N)$ kroków zajmuje zarówno zbudowanie drzewa jak i zamiana drzewa na stóg. Natomiast zbudowanie stogu z każdorazowym przywracaniem własności stogu może w ogólności zająć $O(N \log N)$ kroków.

Jednak ze stogami coś jest nie w porządku. Wyszukiwanie elementu na stogu wypada tak źle jak na drzewach nieuporządkowanych, o których stwierdziliśmy, że z tego powodu nie opłaca się ich używać. Gorzej, elementy na stogach mogą się powtarzać, bo uporządkowanie jest nieostre.

O co tu chodzi?

Klucz do zagadki daje spostrzeżenie, że pewne operacje wypadają dla stogów rewelacyjnie. Na przykład, znajdowanie elementu największego jest zawsze natychmiastowe.

Stogi są po prostu strukturą danych wyspecjalizowaną. Nie nadają się zbyt dobrze do przechowywania elementów z uporządkowaniem według klucza **identyfikującego** dany element, ale świetnie mogłyby się nadać do przechowywania elementów ze względu na ich **ważność** (priorytet). Zwłaszcza jeśli będzie nas interesować tylko znajdowanie elementu o najwyższym priorytecie, i wtedy nawet nie przeszkadza, jeśli priorytety będą się powtarzać. Tego typu strukturę nazywamy kolejką priorytetową.

Abstrakcyjne typy danych

Kolejka priorytetowa jest strukturą bardziej abstrakcyjną niż stóg. Można ją zdefiniować przez podanie zestawu operacji, które chcemy na niej wykonywać:

- wprowadź do struktury nowy element razem z jego priorytetem
- znajdź w strukturze element o najwyższym priorytecie
- usuń ze struktury element o najwyższym priorytecie

Struktura danych, którą definiujemy nie podając jej rzeczywistej struktury, a tylko zestaw operacji, które będą na niej wykonywane, nazywamy *abstrakcyjnym typem danych (ADT)*. ADT są ważnym pojęciem w informatyce, ponieważ wprowadzają pożyteczny poziom abstrakcji, pozwalający specyfikować systemy informatyczne bez decydowania o konkretnych użytych strukturach danych, i zarazem dają programiście wolność wyboru struktury danych implementującej zadany ADT.

Kolejki priorytetowe — specyfikacja ADT

```
FUNCTION UtworzKolejkeP ( VAR K: KolejkaP ):           BOOLEAN; EXTERN;
(* Tworzy i inicjuje kolejke priorytetowa w zmiennej K.        *)
(* Funkcja musi byc wywolana przed pierwszym uzyciem kolejki.  *)
(* Zwraca wartosc TRUE gdy wynik operacji pomyslny.            *)

FUNCTION DodajDoKolejkiP ( VAR K: KolejkaP
                         ; E: TypElementu
                         ; P: TypPriorytetu):          BOOLEAN; EXTERN;
(* Dodaje element E o priorytecie P do kolejke priorytetowa K. *)
(* Zwraca wartosc TRUE gdy wynik operacji pomyslny.            *)

FUNCTION ZnajdzMaxKolejkiP ( K: KolejkaP
                           ; VAR E: TypElementu):      BOOLEAN; EXTERN;
(* Znajduje i zwraca maksymalny element kolejke priorytetowa K.*)
(* Zwraca wartosc TRUE gdy wynik operacji pomyslny.            *)

FUNCTION UsunMaxKolejkiP ( VAR K:KolejkaP ):           BOOLEAN; EXTERN;
(* Usuwa maksymalny element kolejki priorytetowej K.           *)
(* Zwraca wartosc TRUE gdy wynik operacji pomyslny.            *)
```


Kolejki priorytetowe — lepsze sformułowanie

```
FUNCTION DodajDoKolejkiP   ( VAR K: KolejkaP
                           ; E: TypElementu
                           ; FUNCTION wyzszy(e1,e2:TypElementu):BOOLEAN
        ): BOOLEAN; EXTERN;
(* Dodaje element E do kolejki priorytetowej K.                *)
(* Zwraca wartosc TRUE gdy wynik operacji pomyslny.            *)

FUNCTION ZnajdzMaxKolejkiP ( K: KolejkaP
                           ; VAR E: TypElementu
                           ; FUNCTION wyzszy(e1,e2:TypElementu):BOOLEAN
        ): BOOLEAN; EXTERN;
(* Znajduje i zwraca maksymalny element kolejki priorytetowej K.*)
(* Zwraca wartosc TRUE gdy wynik operacji pomyslny.            *)

FUNCTION UsunMaxKolejkiP   ( VAR K: KolejkaP
                           ; FUNCTION wyzszy(e1,e2:TypElementu):BOOLEAN
        ): BOOLEAN; EXTERN;
(* Usuwa maksymalny element kolejki priorytetowej K.           *)
(* Zwraca wartosc TRUE gdy wynik operacji pomyslny.            *)
```


Implementacja kolejek priorytetowych

```
CONST MaxElement = 100;
TYPE WskElementu = ^TypElementu;
     KolejkaP = ^Stog;
     Stog =
        RECORD
          Elementy: ARRAY[0..MaxElement] OF WskElementu;
          Dlugosc: 0..MaxElement;
        END;


FUNCTION UtworzKolejkeP( VAR K: KolejkaP ) : BOOLEAN;
BEGIN
   NEW(K);
   K^.Dlugosc := 0;
   UtworzKolejkeP := TRUE      {tworzenie zawsze sie udaje}
END; {UtworzKolejkeP}
```



```
FUNCTION DodajDoKolejkiP ( VAR K: KolejkaP
                         ; E: TypElementu
                         ; FUNCTION wyzszy(e1,e2: TypElementu):BOOLEAN
        ): BOOLEAN;
BEGIN
  IF K^.Dlugosc >= MaxElement         {sprawdzamy tylko przepelnienie}
  THEN
    BEGIN
      WRITELN('DodajDoKolejkiP -- kolejka pelna');
      DodajDoKolejkiP := FALSE;
    END
  ELSE
    BEGIN
      K^.Dlugosc := K^.Dlugosc + 1;
      NEW(K^.Elementy[K^.Dlugosc]);          {nowy element stogu}
      K^.Elementy[K^.Dlugosc]^ := E;         {kopiujemy caly element}
      PropagacjaWGore(K^,K^.Dlugosc,wyzszy); {przywracamy wlasn.stogu}
      DodajDoKolejkiP := TRUE;
    END
END; {DodajDoKolejkiP}
```

```
FUNCTION ZnajdzMaxKolejkiP ( K: KolejkaP
                           ; VAR E: TypElementu
                           ; FUNCTION wyzszy(e1,e2:TypElementu):BOOLEAN
        ): BOOLEAN;
BEGIN
  {sprawdzamy tylko dlugosc kolejki, nie zas istnienie czy poprawnosc}
  IF K^.Dlugosc <= 0
  THEN
    BEGIN
      WRITELN('ZnajdzMaxKolejkiP -- kolejka pusta');
      ZnajdzMaxKolejkiP := FALSE;
    END
  ELSE
    BEGIN
      E := K^.Elementy[1]^;
      ZnajdzMaxKolejkiP := TRUE;
    END
END; {ZnajdzMaxKolejkiP}
```



```
FUNCTION UsunMaxKolejkiP ( VAR K: KolejkaP
                         ; FUNCTION wyzszy(e1,e2:TypElementu):BOOLEAN
        ): BOOLEAN;
BEGIN
  {sprawdzamy tylko dlugosc kolejki, nie zas istnienie czy poprawnosc}
  IF K^.Dlugosc <= 0
  THEN
    BEGIN
      WRITELN('UsunMaxKolejkiP -- kolejka pusta');
      UsunMaxKolejkiP := FALSE;
    END
  ELSE
    BEGIN
      DISPOSE(K^.Elementy[1]);              {wyrzucamy najstarszy elem}
      K^.Elementy[1] := K^.Elementy[K^.Dlugosc]; {zastepujemy go ostat}
      K^.Dlugosc := K^.Dlugosc - 1;              {skracamy stog}
      PropagacjaWDol(K^,1,wyzszy);          {i przywracamy wlasn.stogu}
      UsunMaxKolejkiP := TRUE;
    END
END; {UsunMaxKolejkiP}
```



```
PROCEDURE PropagacjaWGore(VAR S: Stog; I: INTEGER;
                          FUNCTION wyzszy(e1,e2:TypElementu): BOOLEAN);
{ procedura propaguje I-ty element stogu S w gore  }
{ az osiagnie on wlasciwa sobie pozycje            }
VAR rodzic: INTEGER;
BEGIN
  S.Elementy[0] := S.Elementy[I];
  rodzic := I DIV 2;
  WHILE wyzszy(S.Elementy[0]^,S.Elementy[rodzic]^) DO
    BEGIN
      S.Elementy[I] := S.Elementy[rodzic];
      I := rodzic;
      rodzic := rodzic DIV 2;
    END;
  S.Elementy[I] := S.Elementy[0];
END; {PropagacjaWGore}
```



```
PROCEDURE PropagacjaWDol ( VAR S: Stog; I: INTEGER
                         ; FUNCTION wyzszy(e1,e2: TypElementu): BOOLEAN);
{ procedura propaguje I-ty element stogu S w dol az osiagnie wlasciwa pozycje }
VAR poddrzewo : INTEGER;  koniec: BOOLEAN;
BEGIN
   S.Elementy[0] := S.Elementy[I];
   poddrzewo := 2 * I;
   koniec := FALSE;
   WHILE NOT koniec DO
      IF poddrzewo > S.Dlugosc THEN koniec := TRUE
      ELSE
      BEGIN {wpierw wybieramy poddrzewo o wyzszym priorytecie}
         IF poddrzewo < S.Dlugosc
         THEN IF wyzszy(S.Elementy[poddrzewo+1]^, S.Elementy[poddrzewo]^)
              THEN poddrzewo := poddrzewo + 1;
         {sprawdzamy czy element ma byc zamieniony z poddrzewem}
         IF wyzszy(S.Elementy[0]^, S.Elementy[poddrzewo]^) THEN koniec := TRUE
         ELSE
         BEGIN
            S.Elementy[I] := S.Elementy[poddrzewo];
            I := poddrzewo;
            poddrzewo := 2 * poddrzewo;
      END;
   END;
   S.Elementy[I] := S.Elementy[0];
END; {PropagacjaWDol}
```