VII Algorytmy i struktury danych

4. Struktury danych i ich wpływ na złożoność algorytmów.

Złożoność obliczeniowa

Wybierając konkretny algorytm do rozwiązania problemu, programista powinien wiedzieć, czego może się po nim spodziewać oraz jak jego zastosowanie wpłynie na działanie aplikacji. Często trzeba wybierać spośród kilku możliwych algorytmów i wtedy konieczne jest wprowadzenie jakiegoś kryterium pozwalającego precyzyjnie określić, który z nich jest "najlepszy". Operując na dużych zbiorach danych, najbardziej korzystne jest przyjęcie definicji, w myśl której najlepszy algorytm najefektywniej wykorzystuje dostępne zasoby komputera (pamięć oraz procesor) do jak najszybszego rozwiązania problemu i zaprezentowania wyników. W ten sposób odpowiedzieliśmy sobie na pytanie, czym jest **złożoność obliczeniowa**

Złożoność obliczeniowa jako dział teorii obliczeń zajmuje się określaniem ilości zasobów (np. pamięci, czasu, liczby procesorów) niezbędnych do rozwiązania problemu obliczeniowego.

Pomiar szybkości algorytmu

Czas wykonania danego algorytmu zależy od wielu czynników:

1. Jakości kodu napisanego przez programistę i wygenerowanego przez kompilator
2. Szybkości sprzętu, na którym jest on wykonywany
3. Rozmiaru danych wejściowych
4. Użytego algorytmu

Jakość kodu oraz sprzętu to czynniki czysto subiektywne, które czasem ciężko jest nawet dokładnie określić. Jeśli przyjmiemy, że wszystkie nasze algorytmy badamy na tym samym komputerze i kompilujemy tym samym kompilatorem, oba te czynniki będą mogły zostać wyrażone przez pewną stałą, którą oznaczymy sobie literą *c*. Idźmy dalej - skoro szybkość algorytmu zależy od rozmiaru danych wejściowych, od razu powinno nasunąć to nam podejrzenia, że szybkość algorytmu możemy wyrazić za pomocą pewnej funkcji matematycznej, której argumentem jest rozmiar danych wejściowych *n*. Na przykład w przypadku sortowania, *n* może określać liczbę elementów, które chcemy posortować. Ostateczna postać funkcji zależy od użytego algorytmu.

Przyjęło się, że czas wykonywania algorytmu oznaczany jest przez *T(n)*. Jednostka, w jakiej podawany jest wynik, jest nieważna. Można przyjąć, że jest to liczba instrukcji, jakie musi wykonać komputer. Oto przykład:

$T(n) = cn^2$

c, jak wspomnieliśmy, charakteryzuje tutaj jakość kodu oraz parametry komputera. Wzór ten możemy odczytać następująco: jeżeli liniowo zwiększamy ilość danych, czas wykonywania algorytmu rośnie kwadratowo. Inny algorytm rozwiązujący ten sam problem może być opisany wzorem $T(n) = cn$. Poniżej prezentujemy tabelkę pokazującą, jak wydłuża się czas działania obu algorytmów w zależności od rozmiaru danych wejściowych:

n	1	2	3	4	5	6	7	8	9	10
$T(n) = cn^2$	*c*	*4c*	*9c*	*16c*	*25c*	*36c*	*49c*	*64c*	*81c*	*100c*
$T(n) = cn$	*c*	*2c*	*3c*	*4c*	*5c*	*6c*	*7c*	*8c*	*9c*	*10c*

Z tabelki jasno widać, że drugi z algorytmów cechuje się wyższą wydajnością. Dla danych rozmiaru *7* wykona się on szybciej, niż algorytm pierwszy dla danych rozmiaru *3*. Dla ostatniego argumentu w tabelce - 10 - jest on dziesięciokrotnie szybszy, a im dalej będziemy szli, tym bardziej różnica będzie się pogłębiać.

Okazuje się, że nie zawsze rozmiar jest jedynym czynnikiem, od którego zależy szybkość działania. Niektóre algorytmy bowiem dla pewnych szczególnych danych mogą np. znacząco przyspieszyć, a z kolei inne dane mogą być przetwarzane na nich o wiele dłużej, niż wynika to z naszych obliczeń. W takim przypadku *T(n)* opisuje najgorszy przypadek zwany też *pesymistycznym czasem wykonania* - jest to jednak sytuacja skrajna, wobec czego można także zdefiniować najlepszy możliwy przypadek lub średni czas wykonywania. Nie dajmy się przy tym zwieść pozorom. Stawianie założenia, że wszystkie rodzaje danych są jednakowo prawdopodobne, może prowadzić do nieprzewidzianych rezultatów.

Notacja "dużego O" i "dużej Ω"

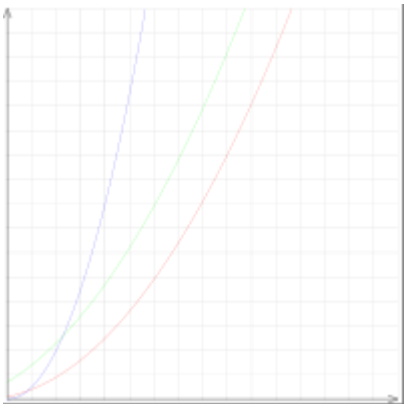
Załóżmy, że mamy dwa algorytmy, o których wiemy, że:

$T_1(n) = c(n + 1)^2$

$T_2(n) = c((n + 3)^2 - 2)$

Oczywiście można tutaj bez większych problemów określić, który z nich jest lepszy, po prostu obliczając wartości obu z nich dla kilku dowolnych argumentów. Jednak na dłuższą metę posługiwanie się takimi wzorami bywa po prostu uciążliwe, nie mówiąc już o niezwykłym zadaniu ich wyznaczenia poprzez studiowanie opisu algorytmu. Przy bardziej rozbudowanych algorytmach sprawa komplikuje się do tego stopnia, że możemy mieć nawet trudności z określeniem jakości każdego z nich. Dlatego do porównywania złożoności algorytmów stosuje się nieco inną notację zwaną notacją "dużego O".

Zapis **T(n) = O(f(n))** oznacza, że istnieją takie stałe dodatnie *c* oraz n_0, że dla każdego $n \geq n_0$ zachodzi $T(n) \leq cf(n)$.



Wykres 1: Notacja "dużego O"

Mówiąc mniej matematycznym językiem, zawsze jesteśmy w stanie znaleźć taką stałą *c* oraz taki argument n_0, że dla wszystkich kolejnych argumentów "czas" wykonywania algorytmu opisanego funkcją T(n) będzie zawsze mniejszy lub równy od "czasu" opisanego funkcją f(n).

Spróbujmy to przenieść na grunt podanych powyżej dwóch funkcji. Dla $T_1(n)$ mamy $T_1(n) = O(n^2)$ oraz $c = 5, n_0 = 1$. Faktycznie, dla każdego n większego lub równego 1 zachodzi nam nierówność $(n+1)^2 \leq 5n^2$. Dla $T_2(n)$ mamy także $T_2(n) = O(n^2)$, ponieważ dla $c = 5$ i $n_0 = 3$ zachodzi nierówność $(n+3)^2 - 2 \leq 5n^2$. Okazuje się zatem, że oba te algorytmy charakteryzują się tym samym rzędem złożoności, zatem dla identycznego rozmiaru danych powinny wykonywać się w podobnym czasie. Funkcja *f(n)* stanowi tutaj górne ograniczenie tempa wzrostu (innymi słowy: algorytm nigdy nie wykona się wolniej), co pokazuje wykres 1. Czerwona krzywa to pierwsza z przedstawionych powyżej funkcji, zielona - druga, a niebieska to funkcja $f(n) = n^2$.

Na oznaczenie dolnej granicy tempa wzrostu (algorytm nigdy nie wykona się szybciej) wprowadzona została notacja "dużej Ω", której definicja jest następująca:

Zapis **T(n) = Ω(g(n))** oznacza, że istnieje taka stała dodatnia *c*, że dla nieskończenie wielu *n* zachodzi $T(n) \geq cg(n)$.

Rząd złożoności

Dobór algorytmu o odpowiedniej złożoności zależy od kilku czynników, m.in. rozmiaru danych oraz mocy komputera. Jeśli mamy dane dwa algorytmy o złożoności sześciennej: $5n^3$ oraz kwadratowej: $100n^2$, to mimo pozornej przewagi drugiego z nich, zauważmy, że dla $n < 20$ działający w czasie sześciennym algorytm okaże się szybszy! Naukowcy zajmujący się algorytmami oraz teorią obliczeń podzielili problemy na kilkanaście klas złożoności. Najważniejsze z nich to:

- *Problemy P* - są to problemy łatwo obliczalne, ich rozwiązanie można znaleźć w czasie wielomianowym (np. n^2, n^6).

- *Problemy NP* - są to problemy trudno obliczalne, ich rozwiązanie można *sprawdzić* w czasie wielomianowym.

Wiadomo, że wszystkie problemy P są też problemami NP, natomiast w drugą stronę sprawa jest nierozstrzygnięta. Jest to jedno z wielkich nierozwiązanych zagadnień matematycznych. Wśród problemów NP można wyróżnić jeszcze kilka podklas. Na szczególną uwagę zasługują problemy NP-zupełne. Ich właściwością jest to, że można do niego w czasie wielomianowym zredukować każdy inny problem NP-zupełny. Znaczy to tyle, że gdyby ktoś znalazł algorytm rozwiązujący w czasie wielomianowym jeden problem NP-zupełny, automatycznie dałoby się w podobnym czasie rozwiązać je wszystkie. Dotychczas jednak wszystkie wymyślone algorytmy mają znacznie większy rząd złożoności, niż większość ludzi jest w stanie zaakceptować, np. $n!$ albo wykładniczy 2^n.

Istnieje też grupa problemów nierozwiązywalnych algorytmicznie.

Rząd złożoności, a moc komputera

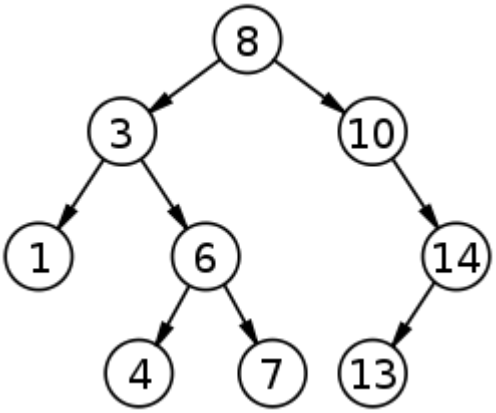
Przełóżmy to teraz na praktykę. Łatwo sprawdzić, że wzrost mocy komputera powoduje znaczący przyrost szybkości jedynie dla algorytmów o niskim rzędzie złożoności, np. liniowym n czy logarytmicznym $n \log n$. Załóżmy, że mamy dwa komputery A i B oraz drugi z nich jest dziesięciokrotnie szybszy od pierwszego. Każdemu z nich dajemy 100 sekund na rozwiązanie czterech problemów algorytmicznych o różnej złożoności i obserwujemy, z jakim rodzajem danych wejściowych poradzi sobie każdy z nich.

Algorytm	**Komputer A**	**Komputer B**	**Przyrost**
$10n$	10	100	10
$4n^2$	5	15	3
n^3	4	10	2,5
2^n	6	10	1,6

Widzimy teraz, że zwiększenie mocy komputera miało najbardziej znaczący wpływ na algorytm liniowy, podczas gdy dla najbardziej zasobożernego algorytmu wykładniczego zaobserwowaliśmy najmniejszy przyrost. Wynika z tego wniosek, że postęp technologiczny ma sens jedynie dla algorytmów o niskim rzędzie złożoności.

Struktury danych

Binarne drzewo poszukiwań



Binarne drzewo poszukiwań o wielkości równej 9, a wysokości równej 3; wierzchołek '8' jest tu korzeniem, a wierzchołki '1', '4', '7' i '13', to liście

Binarne drzewo poszukiwań (skrót BST, z ang. *Binary Search Tree*) – dynamiczna struktura danych w postaci drzewa binarnego stosowana do szybkiego wyszukiwania. Drzewa BST służą do realizacji bardziej abstrakcyjnych struktur danych, np. zbiorów, czy słowników.

W każdym z węzłów drzewa BST przechowywany jest **klucz** (klucze są unikatowe). Wartość klucza jest zawsze **nie mniejsza** niż wartości wszystkich kluczy z lewego poddrzewa, a **nie większa** niż wartości wszystkich kluczy z prawego poddrzewa (relacja może być odwrócona, to kwestia umowy). Relacje niemniejszości i niewiększości można zamienić odpowiednio na relacje większości i mniejszości, ale ograniczamy się wówczas do przypadku unikalności kluczy (wykluczamy klucze, które mogą się powtarzać). Przechodząc drzewo metodą inorder, uzyskuje się ciąg wartości posortowanych niemalejąco (lub rosnąco w przypadku unikatowych kluczy).

Pesymistyczny koszt każdej z operacji na drzewie BST (o liczbie węzłów n), tj. wyszukiwania, dodania lub usunięcia klucza, zależy od wysokości drzewa i wynosi

- $\log_2 n$ – dla drzewa zrównoważonego – najlepszy przypadek;
- n – dla drzewa zdegenerowanego do listy, tj. takiego, w którym każdy z węzłów oprócz liścia ma tylko jednego syna – najgorszy przypadek.

Dlatego w praktyce często lepiej zastosować drzewa BST o dodatkowych właściwościach, np. AVL lub drzewa czerwono-czarne, które poprzez nakładanie pewnych dodatkowych warunków gwarantują, że drzewo będzie zrównoważone (na tyle, na ile to możliwe), dając tym samym logarytmiczny czas dostępu; dzieje się to kosztem większego skomplikowania procedur wstawiających i usuwających klucze.

Działania na BST

Wyszukiwanie klucza

Parametrem wejściowym procedury wyszukującej jest żądana wartość klucza **k** oraz **węzeł** – korzeń drzewa. Algorytm iteracyjny przedstawia się następująco:

1. Jeśli **węzeł** = **nil** (puste poddrzewo) – w drzewie nie ma klucza **k**, przejdź do 5
2. Jeśli klucz[**węzeł**] = **k** – znaleziono węzeł o podanej wartości klucza, przejdź do 5
3. Jeśli klucz[**węzeł**] < **k** – klucz, o ile istnieje w drzewie, zlokalizowany jest w prawym poddrzewie węzła:
    - **węzeł** := prawy_syn[**węzeł**]
    - przejdź do 1
4. Jeśli klucz[**węzeł**] > **k** – klucz, o ile istnieje w drzewie, zlokalizowany jest w lewym poddrzewie węzła:
    - **węzeł** := lewy_syn[**węzeł**]
    - przejdź do 1
5. Koniec wyszukiwania

Wyszukiwanie klucza w drzewie poszukiwań binarnych można też zaimplementować za pomocą returencji:

Szukaj(węzeł):

1. Jeśli **węzeł** = **nil** (puste poddrzewo) – w drzewie nie ma klucza **k**: **return false;**
2. Jeśli klucz[**węzeł**] = **k** – znaleziono węzeł o podanej wartości klucza: **return true;**
3. Jeśli klucz[**węzeł**] < **k** – klucz, o ile istnieje w drzewie, zlokalizowany jest w prawym poddrzewie węzła:
    - **return Szukaj**(prawy_syn[**węzeł**]);
4. Jeśli klucz[**węzeł**] > **k** – klucz, o ile istnieje w drzewie, zlokalizowany jest w lewym poddrzewie węzła:
    - **return Szukaj**(lewy_syn[**węzeł**]);

Wyszukiwanie klucza o minimalnej wartości

1. **x** := korzeń drzewa
2. Dopóki **x** ma lewy następnik:
    - **x** := lewy_syn[**x**]

Po zakończeniu procedury **x** jest węzłem z kluczem o najmniejszej wartości.

Wyszukiwanie klucza o maksymalnej wartości

x **:= korzeń drzewa**

1. Dopóki **x** ma prawy następnik:
    - **x** := prawy_syn[**x**]

Po zakończeniu procedury **x** jest węzłem z kluczem o największej wartości.

Wstawianie klucza

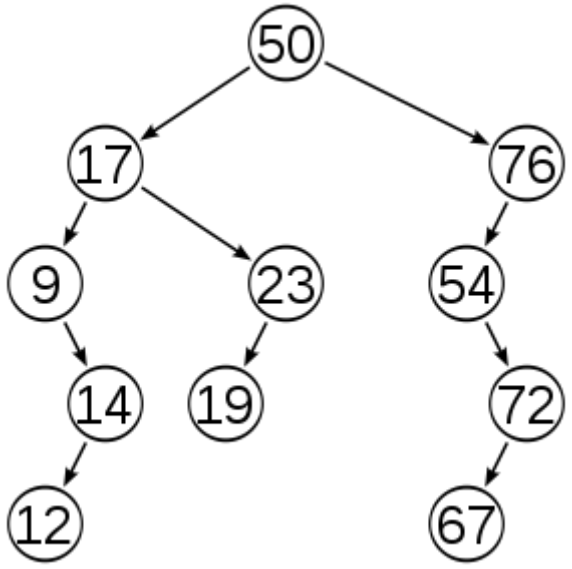
Algorytm jest bardzo podobny jak przy wyszukiwaniu: jeśli przy przechodzeniu drzewa należy przejść w lewo bądź prawo, a węzeł nie ma lewego bądź prawego syna (tj. lewy_syn[**węzeł**] = **nil** lub prawy_syn[**węzeł**] = **nil**), to lewym bądź prawym synem węzła staje się *nowy węzeł*, z kluczem o żądanej wartości. Natomiast jeśli przy przechodzeniu drzewa uda się zlokalizować klucz, to procedura wstawiania nic nie robi.

Usuwanie klucza

Dany jest węzeł **x**, którego klucz ma żądaną wartość. Przy usuwaniu węzła należy rozważyć trzy przypadki:

1. Jeśli **x** jest liściem (nie ma synów) – po prostu usunąć
2. Jeśli **x** ma tylko jednego syna – zastąpić go synem
3. Jeśli **x** ma dwóch synów:
    - Należy znaleźć jego następnik **y** (najbardziej lewy element w prawym poddrzewie **x**)
        - zamiast następnika można też użyć poprzednika (czyli najbardziej prawy element w lewym poddrzewie **x**). Ponieważ nie ma większego znaczenia, którego z nich użyjemy, można zastosować metodę naprzemienną, albo nawet losową. Logiczne jest również używanie zawsze poprzednika lub zawsze następnika.
    - Zastąpić zawartość węzła **x** przez zawartość węzła **y**
    - Wyciąć węzeł **y** o którym wiemy, że może posiadać najwyżej jednego - prawego - syna (ewentualny syn **y** stanie się synem swojego dziadka)

Wyważanie drzewa



Drzewo niezrównoważone (np. lewe poddrzewo węzła 76 ma wysokość 3, a prawe 0)

Czas wykonywania operacji na drzewach poszukiwań binarnych zależy od średniej wysokości drzewa. Najlepiej, gdy wynosi ona w przybliżeniu log*n*, gdzie *n* to ilość węzłów w drzewie.

Jest tak w przypadku, gdy drzewo jest **zrównoważone**, wówczas różnica wysokości lewego i prawego poddrzewa każdego z węzłów wynosi co najwyżej 1. Drzewo jest **doskonale zrównoważone**, gdy dodatkowo wszystkie liście znajdują się na najwyżej dwóch poziomach.

Jak wspomniano na wstępie drzewa AVL i czerwono-czarne gwarantują zrównoważenie, kosztem jednak większej złożoności operacji wstawiania i usuwania.

Wyważone drzewo można łatwo zbudować na bazie posortowanej tablicy, algorytmem analogicznym do wyszukiwania binarnego. Jednak to podejście jest nie najlepsze jeśli drzewo już istnieje – wymaga bowiem skopiowania wszystkich danych do dodatkowej pamięci i całkowitej przebudowy drzewa. Lepszym rozwiązaniem jest użycie algorytmu DSW, który równoważy drzewo BST poprzez szereg rotacji, co nie wymaga ani sortowania, ani dodatkowej pamięci.

Optymalne drzewo poszukiwań

Jeśli wiadomo, że dane w drzewie nie będą zmieniane, a ponadto znane są prawdopodobieństwa (częstotliwości) dostępu do poszczególnych kluczy, można utworzyć optymalne BST, w którym **oczekiwany czas wyszukiwania** będzie minimalny. Przy konstrukcji drzewa optymalnego bierze się pod uwagę dwa czynniki: prawdopodobieństwo wyszukiwania zakończonego powodzeniem (klucz jest w drzewie) oraz niepowodzeniem (klucza nie ma w drzewie).

W przypadku nieznanej statystyki dostępu do kluczy można stosować drzewa samoorgnizujące się, które na bieżąco korygują położenie kluczy w drzewie.

Implementacja

Przykładowy program w języku Pascal. Buduje drzewo BST i przekształca je w drzewo wyważone.

```
program BST;

uses crt;

type
        wsk     =       ^wezel;
        wezel   =       record
                                d : integer;
                                l : wsk;
                                r : wsk;
                        end;

var
        n,i,x   : integer;
        p       : wsk;

procedure Wstaw(var p : wsk; x : integer);
begin
        if p = nil then
        begin
                new(p);
                p^.d := x;
```

```
                p^.l := nil;
                p^.r := nil;
        end
        else if x<p^.d then
                Wstaw(p^.l,x)
        else
                Wstaw(p^.r,x);
end;

function Licz(p : wsk) : integer;
var k : integer;
begin
        if p <> nil then
        begin
                k := 1;
                if p^.l <> nil then
                        k := k+Licz(p^.l);
                if p^.r <> nil then
                        k := k+Licz(p^.r);
                Licz := k;
        end
        else Licz := 0;
end;

procedure Pokaz(p : wsk);
begin
        writeln(p^.d);
        if p^.l <> nil then Pokaz(p^.l);
        if p^.r <> nil then Pokaz(p^.r);
end;

procedure Wywaz(var p : wsk; b : integer);
var a : integer; q, w : wsk;
begin
        b := b-1;
        a := Licz(p^.l);
        b := b-a;
        while abs(a-b) > 1 do begin
                if a > b then begin
                        if p^.l^.r <> nil then begin
                                q := p^.l;
                                repeat
                                        w := q;
                                        q := q^.r;
                                until q^.r = nil;
                                q^.r := p;
                                q^.l := p^.l;
                                w^.r := nil;
                                p^.l := nil;
                                p := q;
                        end
                        else begin
                                p^.l^.r := p;
                                q := p^.l;
                                p^.l := nil;
                                p := q;
                        end;
                        a := a-1;
                        b := b+1;
                end
                else begin
```

```
                        if p^.r^.l <> nil then begin
                                q := p^.r;
                                repeat
                                        w := q;
                                        q := q^.l;
                                until q^.l=nil;
                                q^.l := p;
                                q^.r := p^.r;
                                w^.l := nil;
                                p^.r := nil;
                                p := q;
                        end
                        else begin
                                p^.r^.l := p;
                                q := p^.r;
                                p^.r := nil;
                                p := q;
                        end;
                        a := a+1;
                        b := b-1;
                end;
        end;
        if p^.l <> nil then Wywaz(p^.l,a);
        if p^.r <> nil then Wywaz(p^.r,b);
end;

begin
        p := nil;
        clrscr;
        writeln;
        write('Ile elementow? ');
        readln(n);
        for i := 1 to n do begin
                write('Element numer ',i,': ');
                readln(x);
                Wstaw(p,x);
        end;
        writeln;
        writeln('Nie wywazone:');
        Pokaz(p);
        n := Licz(p);
        writeln('Elementow jest ',n);
        Wywaz(p,n);
        writeln;
        writeln('Wywazone:');
        Pokaz(p);
        readln;
end.
```

Sposoby przechodzenia drzewa binarnego

Istnieje 6 sposobów przejścia drzewa binarnego: VLR, LVR, LRV, VRL, RVL, RLV, gdzie: **V**isit - "odwiedź" węzeł, **L**eft - idź w lewo, **R**ight - idź w prawo. Wyróżnia się 3 pierwsze:

- VLR - **pre-order**, przejście wzdłużne
- LVR - **in-order**, przejście poprzeczne

- LRV - **post-order**, przejście wsteczne

W przypadku gdy dane drzewo jest binarnym drzewem AST przejścia określa się również:

- pre-order - **prefiksowym**, gdyż wynik odwiedzania poszczególnych węzłów jest trawestacją wyrażenia zawartego w strukturze AST do postaci przedrostkowej (notacji Łukasiewicza)
- in-order - **infiksowym**, gdyż trawestuje wyrażenie do postaci wrostkowej
- post-order - **postfiksowym**, gdyż trawestuje wyrażenie do postaci przyrowstkowej

Podane algorytmy rekurencyjne działają na drzewie binarnym :

- **Pre-order**

```
PRE-ORDER(wierzchołek_v)
 {
    wypisz wierzchołek_v.wartość
    jeżeli wierzchołek_v.lewy_syn != null to PRE-
ORDER(wierzchołek_v.lewy_syn)
    jeżeli wierzchołek_v.prawy_syn != null to PRE-
ORDER(wierzchołek_v.prawy_syn)
 }
```

Działanie jest wykonywane najpierw na rodzicu, następnie na synach.

- **In-order**

```
IN-ORDER(wierzchołek_v)
 {
    jeżeli wierzchołek_v.lewy_syn != null to IN-
ORDER(wierzchołek_v.lewy_syn)
    wypisz wierzchołek_v.wartość
    jeżeli wierzchołek_v.prawy_syn != null to IN-
ORDER(wierzchołek_v.prawy_syn)
 }
```

Najpierw wykonywane jest działanie na jednym z synów, następnie na rodzicu i na końcu na drugim synu. Przechodząc w ten sposób drzewo poszukiwań binarnych, otrzymuje się posortowane wartości wszystkich węzłów. Dzieje się tak dlatego, że w drzewie poszukiwań binarnych wartości lewego syna węzła *n* oraz wszystkich jego potomków są mniejsze od wartości *n*, a wartości prawego syna i jego potomków większe od wartości *n*.

- **Post-order**

```
POST-ORDER(wierzchołek_v)
 {
    jeżeli wierzchołek_v.lewy_syn != null to POST-
ORDER(wierzchołek_v.lewy_syn)
    jeżeli wierzchołek_v.prawy_syn != null to POST-
ORDER(wierzchołek_v.prawy_syn)
    wypisz wierzchołek_v.wartość
 }
```

Działanie jest wykonywane najpierw na wszystkich synach, na końcu na rodzicu.

Kopiec binarny

Kopiec binarny (czasem używa się też określenia *sterta*) (ang.*binary heap*) - kopiec stworzony jako drzewo binarne.

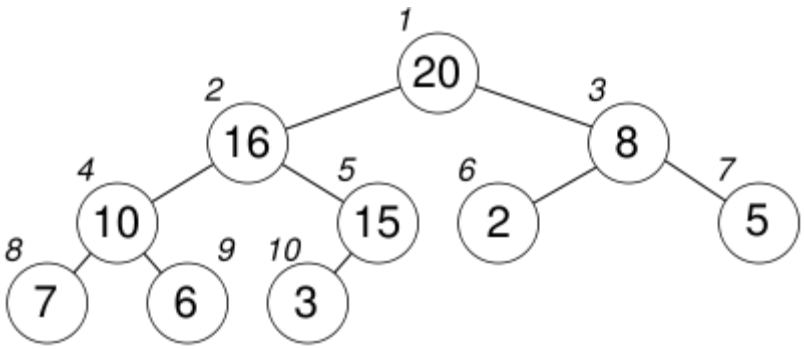
Kopiec binarny musi być **kopcem zupełnym**, czyli

- liście występują na ostatnim i ewentualnie przedostatnim poziomie w drzewie (gdy ostatni poziom nie jest całkowicie wypełniony) ,
- liście na ostatnim poziomie są spójnie ułożone od strony lewej do prawej.

Przechowywanie w pamięci

Kopiec binarny bardzo łatwo przechowywać w pamięci. Zazwyczaj zapisuje się go w tablicy. Zauważmy, że każdy kolejny poziom kopca *i* zawiera 2^i wierzchołków, z wyjątkiem poziomu ostatniego *n*, który może mieć od 1 do 2^n wierzchołków.

Na rysunku kursywą zapisano indeksy w tablicy.



Korzeń kopca binarnego znajduje się w tablicy pod indeksem *1*, jego dzieci są pod indeksami *2* i *3*. Dzieci elementu nr *2* mają indeksy *4* i *5*, a elementu *3 – 6* i *7*. W ten sposób powyższy kopiec można przedstawić jako następującą tablicę:

Indeks	1	2	3	4	5	6	7	8	9	10
Klucz	20	16	8	10	15	2	5	7	6	3

Ogólnie:

- dzieci wierzchołka o indeksie *i* mają indeksy:

  2*i* - dziecko lewe
  2*i*+1 - dziecko prawe

- rodzic wierzchołka o indeksie i>1 ma indeks równy:

  $$\left\lfloor \frac{i}{2} \right\rfloor$$

  operację tę można wykonać przesunięciem arytmetycznym w prawo - bardzo szybką operacją dostępną na praktycznie każdym mikroprocesorze.

Budowanie i naprawianie struktury kopca w jezyku C:

```
// naprawianie struktury kopca
void heapify (int a)
{
        int largest = a;
        if ( (2 * a <= size ) && ( H[2 * a] > H[largest] ) )
                largest = 2 * a;
        if ( (2 * a + 1 <= size ) && ( H[2 * a + 1] > H[largest] ) )
                largest = 2 * a + 1;
        if ( largest != a)
        {
                swap( &H[largest], &H[a]);
                heapify(largest);
        }
}

// budowanie kopca
void build_heap()
{
        for (int i = size; i >= 1; i--)
                heapify(i);
}
```

Dodawanie nowych wierzchołków

Załóżmy, że kopiec składa się z n elementów, zaś elementy uporządkowane są od największych (warunek kopca brzmi więc: każdy element jest większy od swoich dzieci). Dodawany wierzchołek ma klucz równy k:

1. wstaw wierzchołek na pozycję n+1
2. zamieniaj pozycjami z rodzicem (przepychaj w górę) aż do przywrócenia warunku kopca (czyli tak długo, aż klucz rodzica jest większy niż k, lub element dotrze na pozycję 1)

Dodawanie nowego wierzchołka w języku C:

```
void insert (int a)
{
        size++;
        H[size] = a;
        int child = size;
        while ( H[child] > H[child/2] )
        {
                swap( &H[child], &H[child/2] );
                child /= 2;
                if ( child == 1 ) break;
        }
}
```

Usuwanie wierzchołka ze szczytu kopca

1. usuń wierzchołek ze szczytu kopca
2. przestaw ostatni wierzchołek z pozycji n+1 na szczyt kopca; niech k oznacza jego klucz
3. spychaj przestawiony wierzchołek w dół, zamieniając pozycjami z większym z dzieci, aż do przywrócenia warunku kopca (czyli aż dzieci będą mniejsze od k lub element dotrze na spód kopca)

Zarówno wstawianie jak i usuwanie obiektów ze szczytu kopca ma złożoność $O(\log n)$.