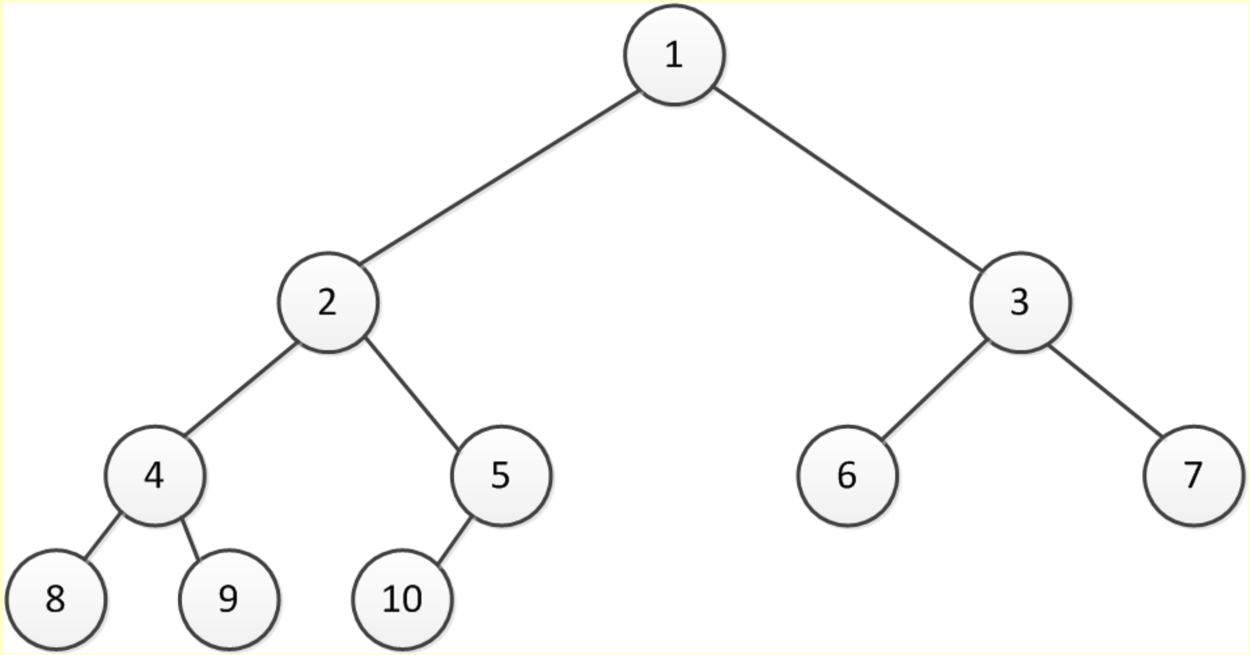
Algorytmy sortownia przez kopcowanie

Sortowanie przez kopcowanie binarne

W celu zmniejszenia złożoności obliczeniowej sortowania bąbelkowego należy zauważyć, że niektóre porównania wykonywane w trakcie realizacji algorytmu są zbędne. Drzewo binarne pełne pomoże nam w wyeliminowaniu niepotrzebnych porównań w trakcie algorytmu sortowania bąbelkowego.

Drzewo binarne to drzewo, w którym każdy z węzłów ma dwa węzły potomne, poza ostatnim rzędem tzw. liści. Drzewo pełne oznacza, ze nie może brakować węzła wewnątrz drzewa i jednocześnie wszystkie liście są dosunięte do lewej strony tak jak pokazano na poniższym rysunku. Numeracja w drzewie rozpoczyna się od 1 do *n* i przechodzi od znajdującego się na samym szczycie węzła poprzez wszystkie kolejne poziomy. Jak widać na kolejnej ilustracji numerowanie zaczynamy z lewej strony każdego poziomu i takie postępowanie kontynuujemy aż przejdziemy wszystkie poziomy na liściach drzewa kończąc.

Kopiec poddany rotowaniu



W ten sposób tworzymy układ kopca, który ma swoją odpowiednią definicję w matematyce. Kopcem nazywamy graf dla $i = \frac{n}{2}, \ldots, 0$ spełniający następujące warunki:

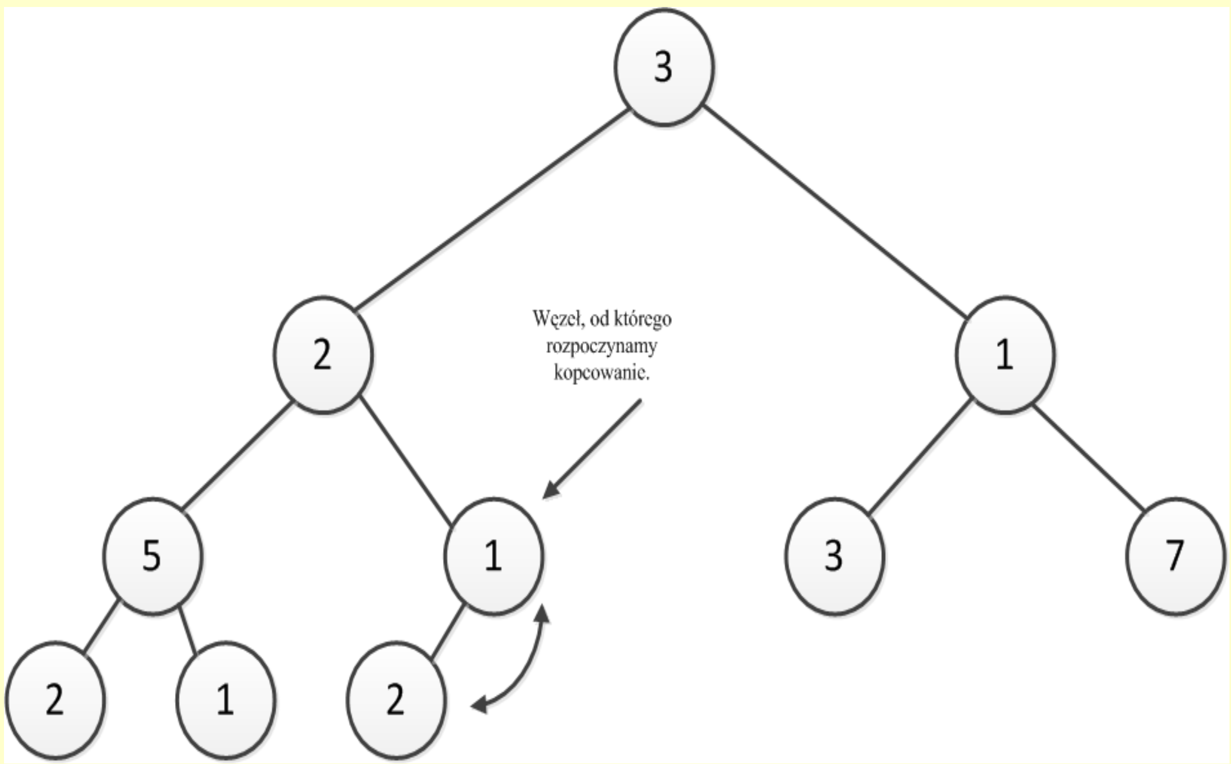
$$a[i] \geq a[2 \cdot i]$$

oraz

$$a[i] \geq a[2 \cdot i + 1]$$

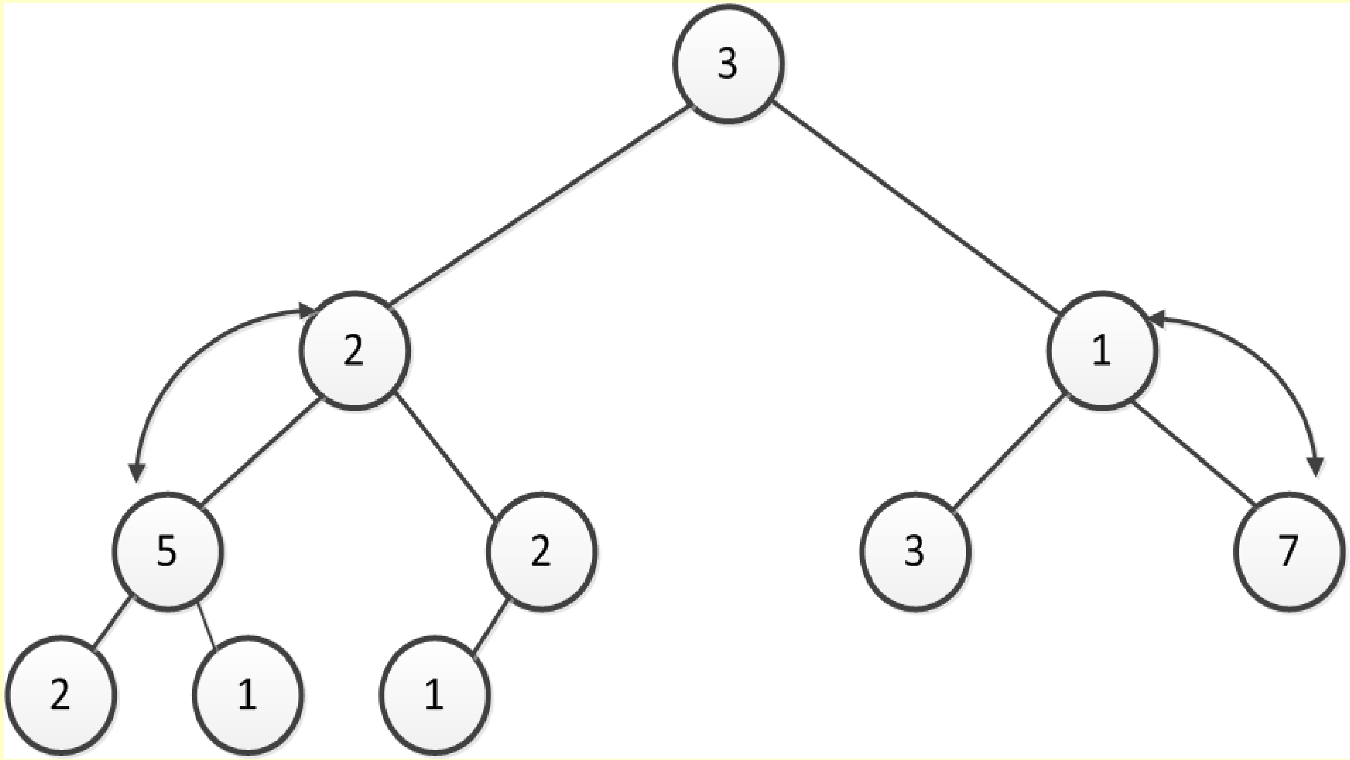
Kopiec z pierwszym węzłem – początek sortowania przez kopcowanie binarne

Zapis powyższy oznacza, że każdy węzeł przechowuje element większy od elementów w węzłach potomnych. W naszym przykładzie n=10 i jednocześnie przyjęto następujący ciąg liczb do posortowania: 3,2,1,5,1,3,7,2,1,2. Elementy te wpisujemy w kolejne węzły grafu zgodnie z numerami. Algorytm kopcowanie rozpoczynamy od węzła o numerze $\left\lfloor \frac{n}{2} \right\rfloor = \left\lfloor \frac{10}{2} \right\rfloor$=5.



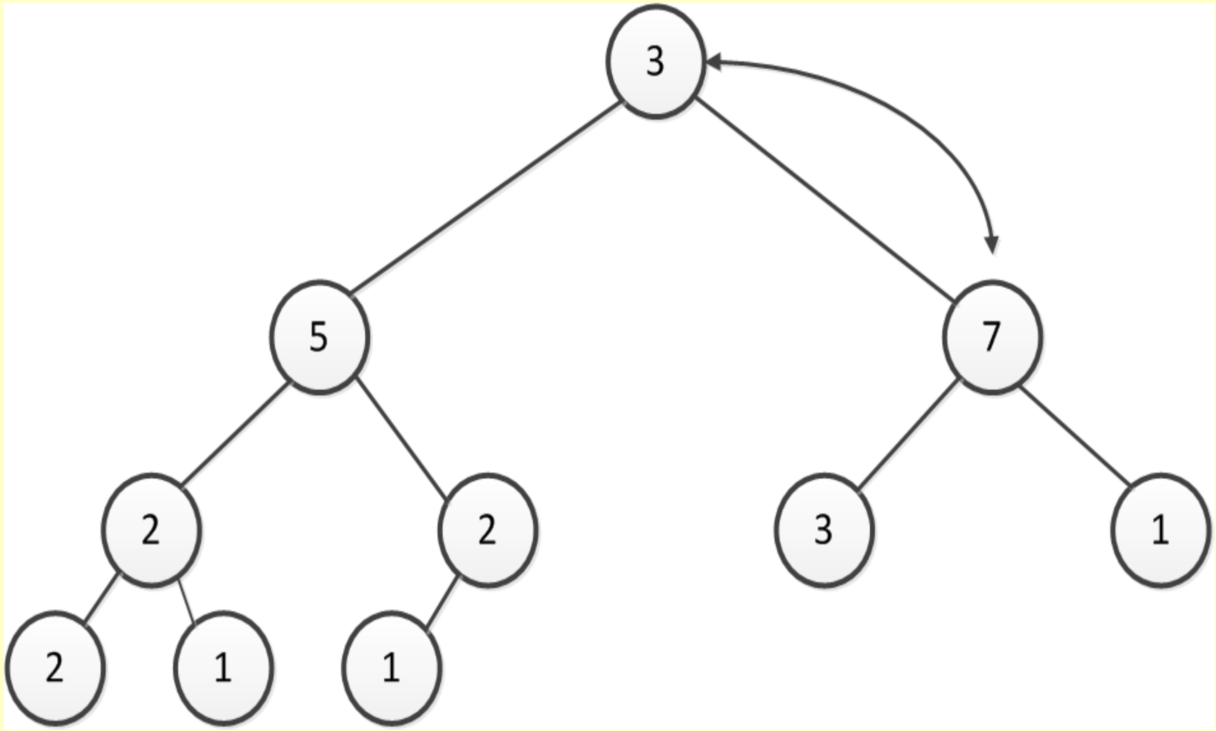
2 etap sortowania przez kopcowanie binarne

Ponieważ węzeł potomny przechowuje wartość większą od węzła nadrzędnego zamieniamy wartości w węzłach. Następnie przechodzimy do węzła o numerze 4, który spełnia warunek kopcowania. Przesuwanie elementów w kolejnych dwóch węzłach zostało pokazane na rysunku.



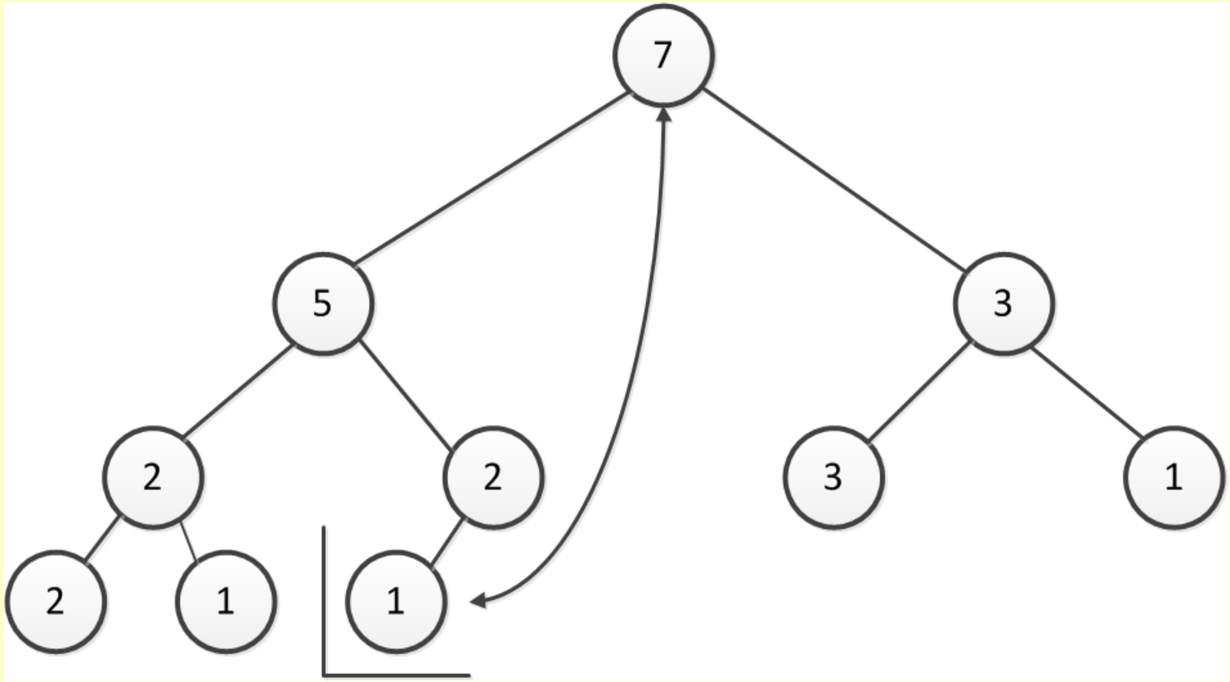
3 etap sortowania przez kopcowanie binarne

W wyniku przestawień elementów otrzymujemy ciąg liczb: 3,2,1,5,2,3,7,2,1,1. W ostatnim kroku przechodzimy do korzenia drzewa i dokonujemy przestawienia tak, aby na górze znajdował się element największy. Taki stan uzyskamy poprzez zamianę 3 znajdującej się w korzeniu z 7 znajdującą się w węźle oznaczonym numerem 3. Operacja ta została pokazana na poniższej ilustracji.



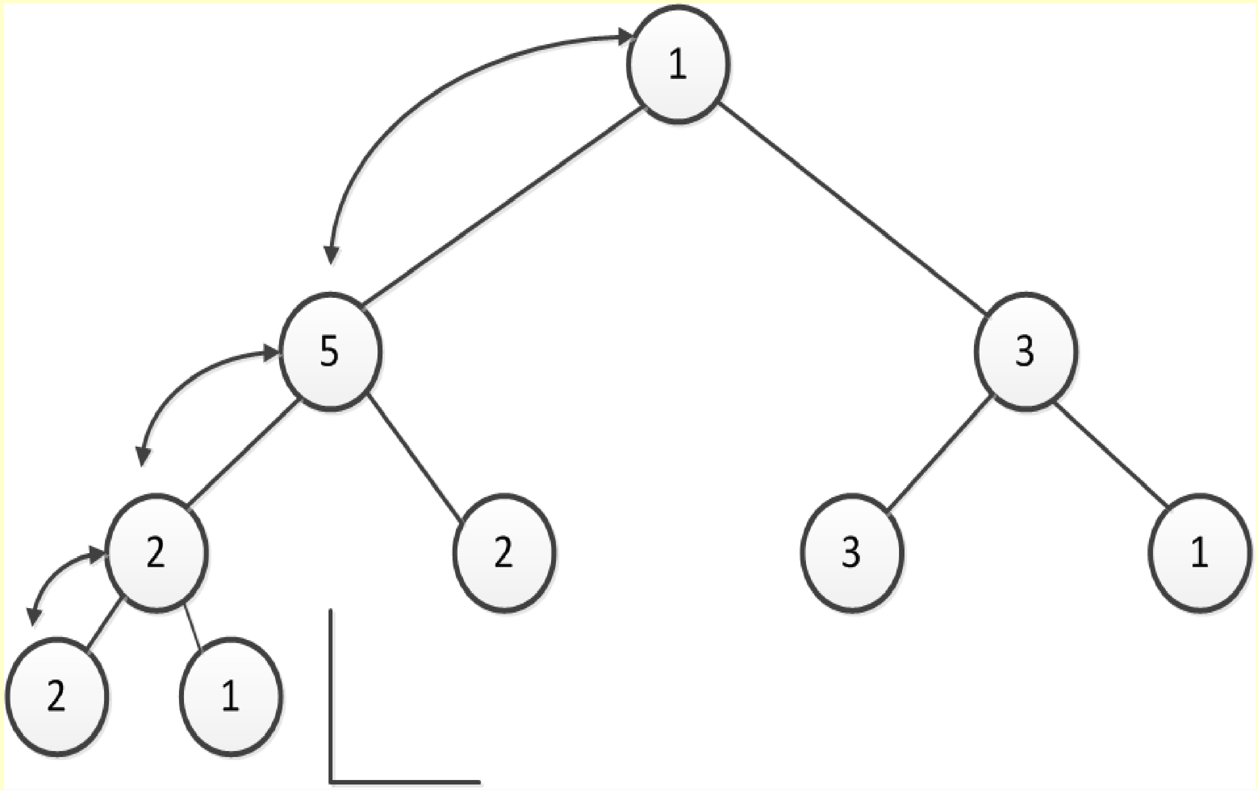
4 etap sortowania przez kopcowanie binarne

W wyniku przesuwania elementów otrzymujemy kopiec spełniający warunki kopca dla i=1,…,n/2. Obecnie nasz ciąg liczbowy jest następujący: 7,5,3,2,2,3,1,2,1,1. Istotne jest stwierdzenie faktu, że kopcowania możemy dokonać w czasie $\vartheta(n)$. Ponieważ największy element znajduje się na początku ciągu przestawiamy go z elementem ostatnim i dokonujemy kopcowania z wyłączeniem elementu ostatniego tak samo jak w przypadku sortowania bąbelkowego.



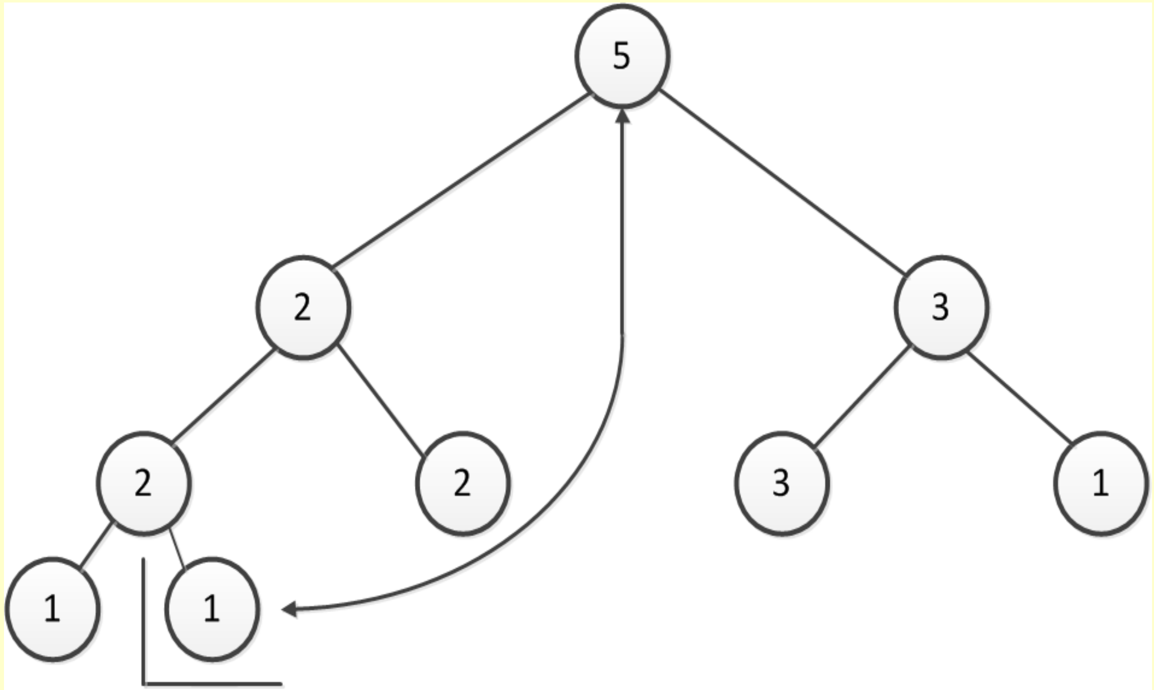
5 etap sortowania przez kopcowanie binarne

Po przestawieniu dokonujemy przesunięcia elementu przechowywanego w korzeniu drzewa w dół drzewa kopcowany ciąg liczbowy jest następujący: 1,5,3,2,2,3,1,2,1,1,7. Element, który podlega zamianie z korzeniem drzewa został specjalnie zaznaczony na powyższej ilustracji.



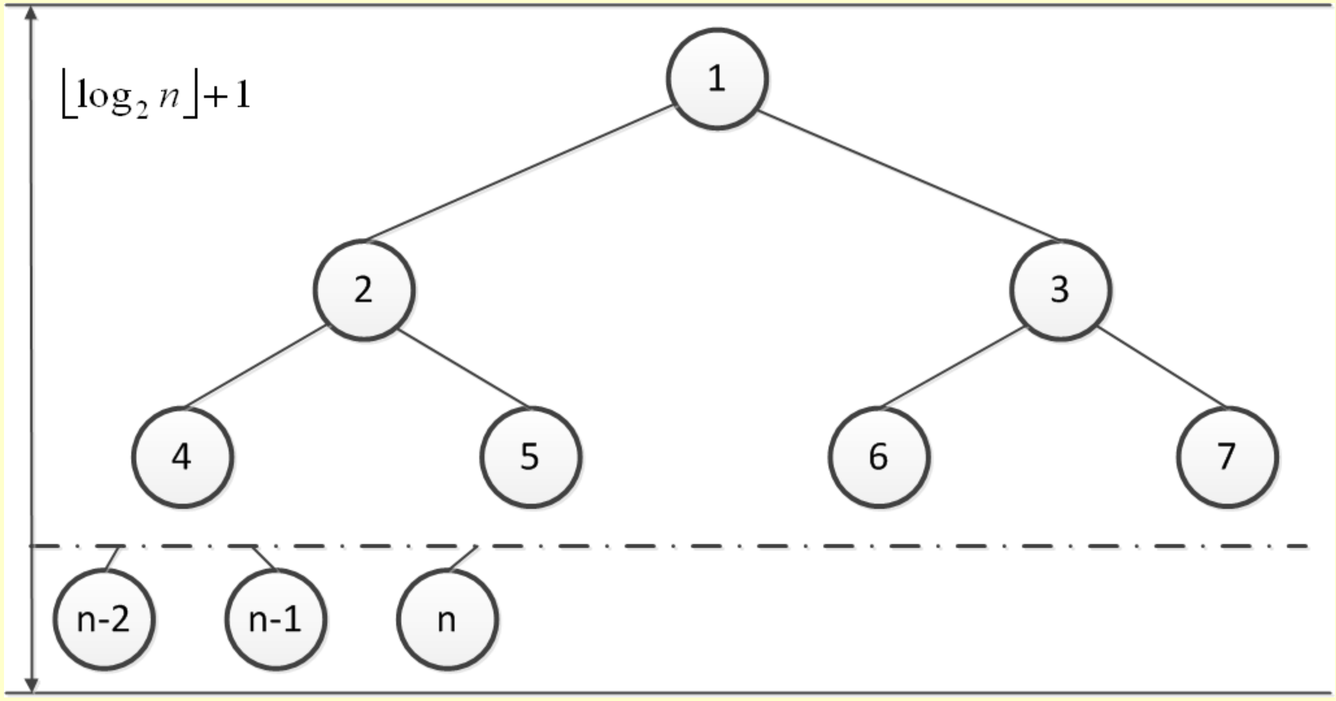
6 etap sortowania przez kopcowanie binarne

Po przestawieniu ciąg liczbowy jest następujący: 1,5,3,2,2,3,1,2,1. Nie wyróżniamy już w nim elementu największego znajdującego się na końcu drzewa w liściu oznaczonym numerem 10. W miejscu największego elementu w całym drzewie pozostawiono puste oznaczenie, ponieważ ten element ciągu nie będzie podlegał dalszym przestawieniom. Następnie dokonujemy przestawienia elementu największego w obecnym ciągu na pozycję przedostatnią w wyjściowym ciągu. Oznacza to, że przestawiamy 5 z korzenia na ostatni element podobnie jak to było w poprzednim kroku dla największej liczby 7, która teraz pozostaje bez zmian.



Schemat dla n wymiarowego kopca

Po wykonaniu kolejnych kroków otrzymujemy uporządkowany ciąg liczb od najmniejszej do największej. Dla ustalenia ilości wykonanych operacji wystarczy zauważyć, że wysokość pełnego drzewa binarnego wynosi $\lfloor \log_2 n \rfloor + 1$, co pokazano na poniższej ilustracji. Wynika to bezpośrednio z definicji funkcji logarytmicznej. Symbol $\lfloor\ \rfloor$ jest rozumiany jako zaokrąglenie do najmniejszej liczby całkowitej w dół. Stąd czas działania algorytmu sortowania przez kopcowania wynosi $\vartheta(n \log_2 n)$.



Przykład 48

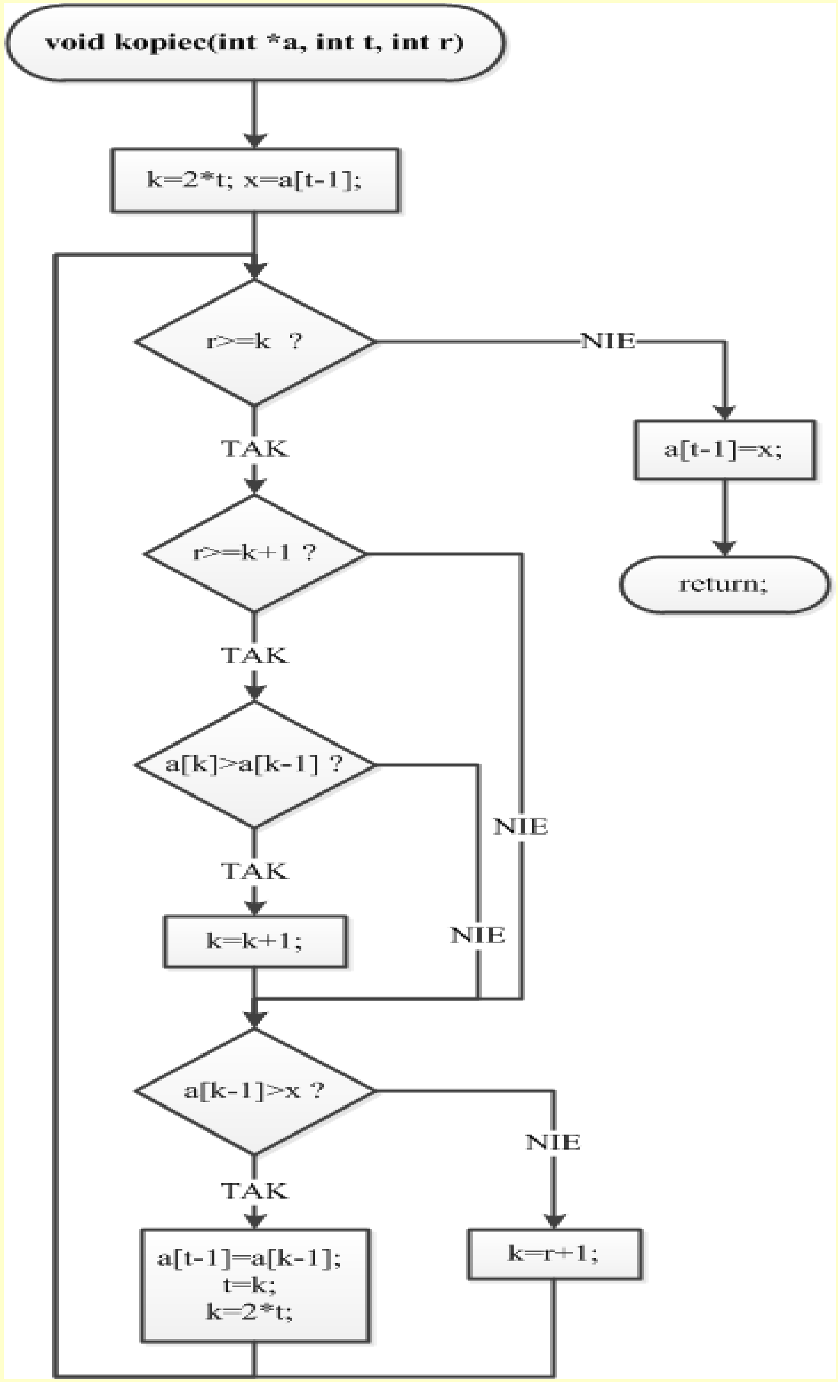
Implementacja prezentowanego algorytmu sortowania poprzez kopcowanie binarne.

```
#include<stdio.h>                          // Deklaracja biblioteki
void kopiec(int  *a,  int  t, int   r)     /* Deklaracja funkcji budującej kopiec */
{
   int  x, k;
   k = 2*t;
   x = a[t-1];
   while(r >= k)                           /* Procedura wypełniania poszczególnych miejsc kopca zgodnie z przyjętymi wartościami. */
   {
      if(r >= k + 1)
      if(a[k] > a[k-1]) k = k + 1;
      if(a[k-1] > x)
                  {   a[t-1] = a[k-1];
                      t = k;
                      k = 2*t;   }
    else k = r + 1;
   }
   a[t-1] = x;
   return;
}
```

``void kopcowanie(int *a, int n)```	// Deklaracja funkcji sortującej kopiec
``{``` ``` int i,x;``` ``` for(i = (n/2); i > 0; i--)``` ``` kopiec(a,i,n);```	/* Wywołanie tworzenia kopca. Należy zwrócić uwagę, że poszczególne funkcje mogą wywoływać inne. */
`` for(i = n-1; i > 0; --i)``` ``` {``` ``` x = a[i];``` ``` a[i] = a[0];``` ``` a[0] = x;``` ``` kopiec(a ,1 ,i);``` ``` }``` ``` return;``` ```}``` ```int main()``` ```{``` ``` int i, n=20,```	/* Zamiana miejscami odpowiednich elementów kopca. Następnie wywołujemy funkcje tworzącą kopiec dla nowego ciągu elementów. */
`` a[]={19,18,17,16,15,14,13,12,11,10,9,8,7,6,``` ``` 5,4,3,2,1,0};```	/* Ciąg poddany kopcowaniu. */
`` kopcowanie(a,n);```	/* Wywołanie sortowania kopca */
`` for(i = 0; i < n; i++)``` ``` printf("a[%d]=%d\n",i,a[i]);``` ```}```	/* Wypisanie poukładanego kopca */

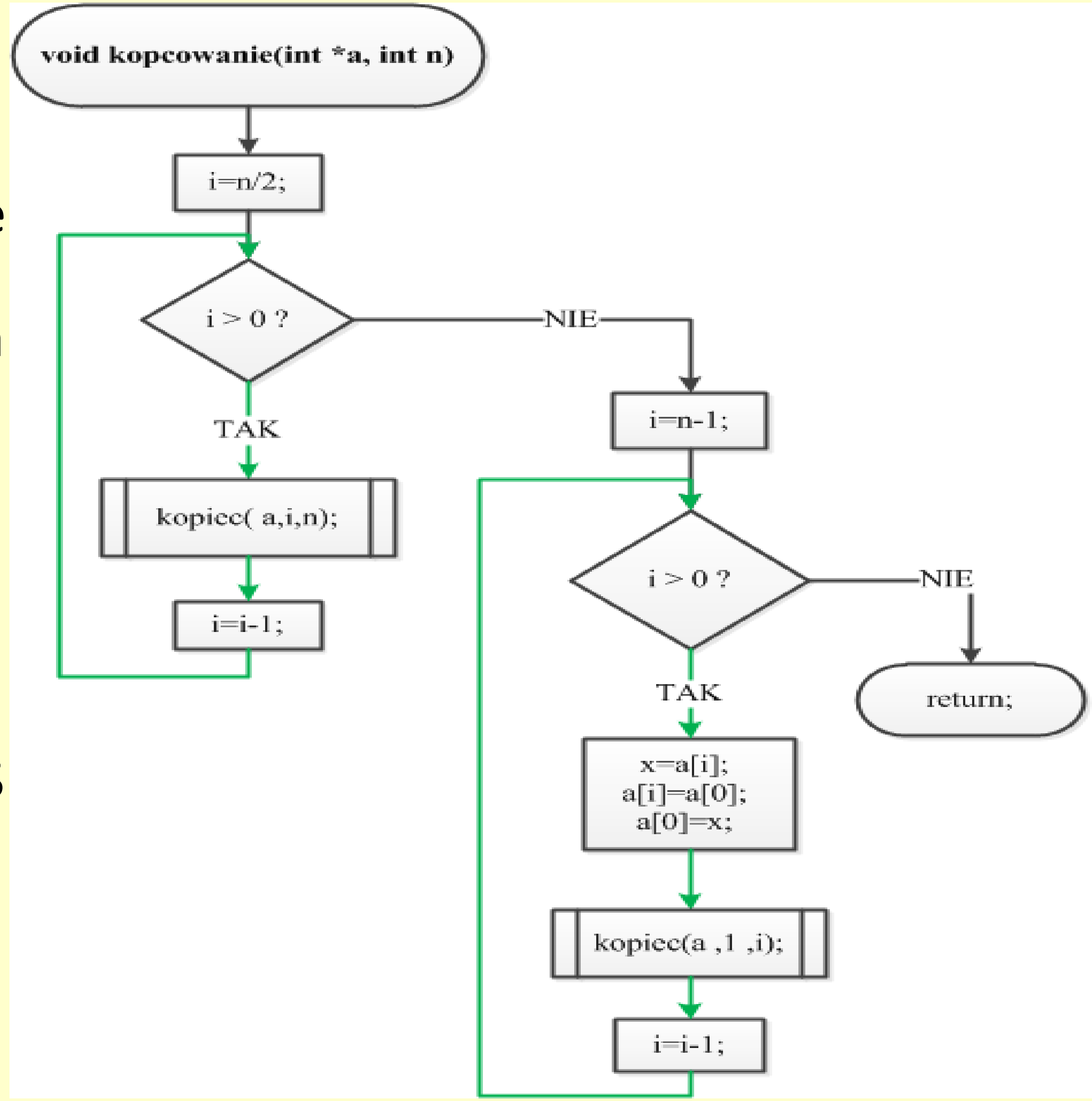
- Schemat blokowy procedury budowy kopca .

Czytelnik zechce prześledzić jeszcze raz omawiane procedury na kolejnych schematach blokowych. Dla sortowania przez kopcowanie binarne zostały przedstawione dwa schematy blokowe. Jedne opisuje sposób, w jaki będziemy budowali kopiec w każdej iteracji algorytmu sortującego. Natomiast drugi pokazuje procedurę kocowania rozumianą jako układanie elementów na stworzonym kopcu (drzewie pełnym binarnym). Czytelnik zechce porównać przedstawione ilustracje z kodem programu. Opisane procedury w trakcie realizacji programu wywołują się wzajemnie przekazując sobie dane. Dzięki implementacji programu w postaci kilku powiązanych procedur możemy jego elementy łatwiej przenosić lub modyfikować.

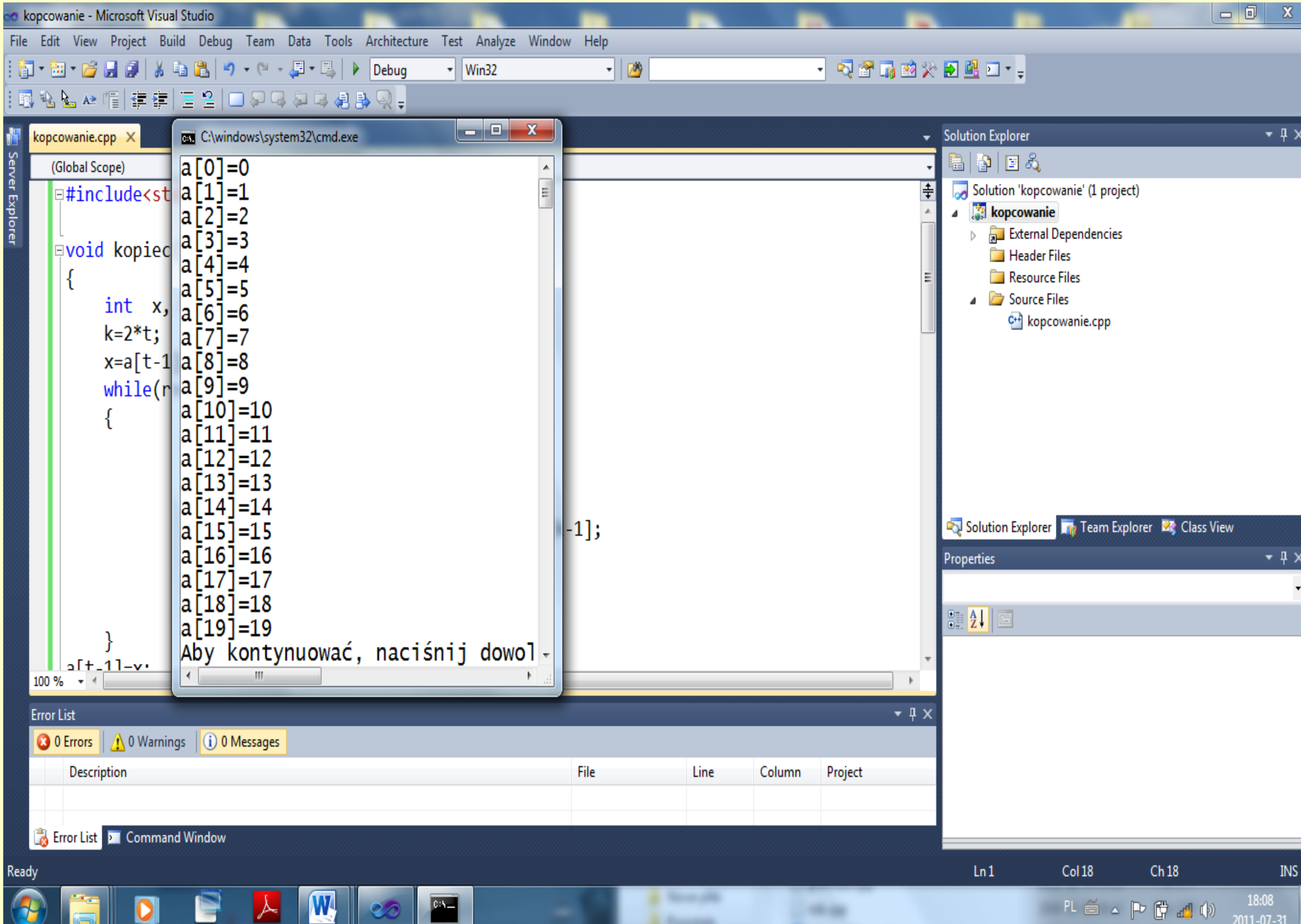


- Schemat blokowy sortowania przez kopcowanie binarne

Opisana powyższym schematem procedura budowy kopca jest wywoływana poprzez procedurę kocowania, opisaną na kolejnej ilustracji, w trakcie realizacji algorytmu sortującego. Kompilacja kodu w MS Visual Studio została przedstawiona na kolejnych ilustracjach



Okno kompilacji przykładu sortowania przez kopcowanie binarne w MS Visual Studio

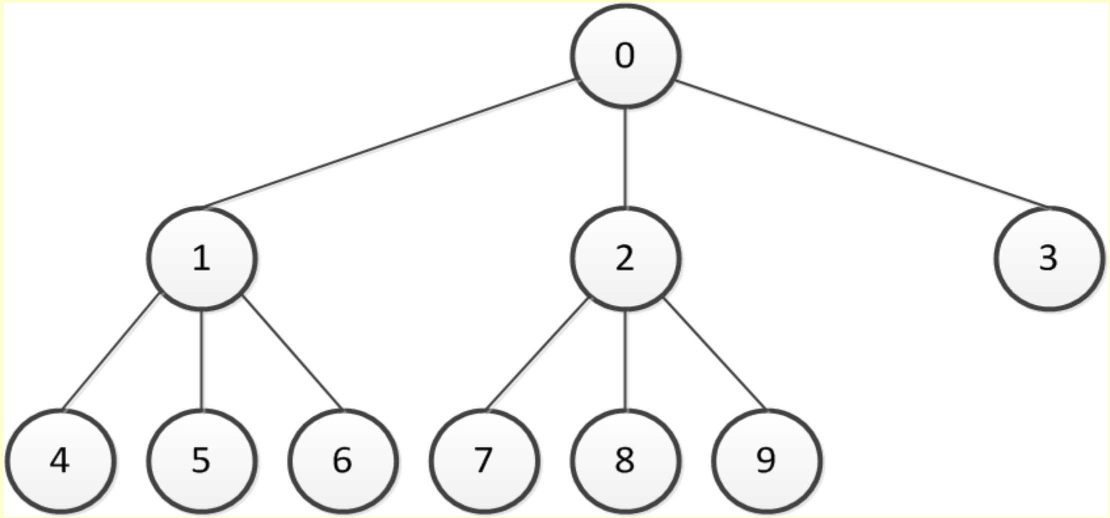


Sortowanie przez kopcowanie trójdzielne

Programowanie procedur i algorytmów jest w dużej mierze zależne od pomysłu na rozwiązanie zadanego problemu. Bardzo często możemy zmodyfikować zbudowaną już procedurę, aby lepiej dopasować jej działanie do danego zadania. Najczęściej jednak staramy się projektować nową metodę o krótszej złożoności obliczeniowej lub czasowej wykonywanych operacji. Uzyskanie zmniejszenia tych wartości dla nowego schematu metody pozwalają na oszczędność czasu lub zmniejszenie ilości miejsca w pamięci komputera jak jest wymagana do realizacji zadania. W literaturze [1-2, 33, 37, 39, 45-47, 51, 67, 69, 71, 73, 82, 85, 90, 91, 100, 107, 112, 114, 115] możemy znaleźć wiele ciekawy modyfikacji omawianych metod sortowania.

Spróbujmy teraz zmniejszyć złożoność czasową algorytmu sortowania przez kopcowanie. W tym celu rozpatrzymy drzewo trójdzielne dla schematu kopca złożonego z 10 węzłów. Numeracja węzłów w tym schemacie drzewa rozpoczyna się od zera, a kończy na węźle o numerze *n*-1, co zostało pokazane na poniższej ilustracji. Modyfikacja metody polega na wprowadzeniu drzewa trójdzielnego w miejsce dwudzielnego, co pozwoli szybciej wykonywać niektóre operacje algorytmu sortującego. Prześledźmy proponowana modyfikację na przykładzie ilustrującym jej działanie

Kopiec trójdzielny



Wstawmy w węzły grafu ciąg liczb: 3,2,1,5,2,3,7,2,1,1. Rozpoczniemy procedurą kopcowania od węzła o numerze $\frac{n-2}{3}$. Kopiec trójdzielny dla $i = \frac{n-2}{3}, \ldots, 0$ spełnia warunki

$$a[i] \geq a[3 \cdot i + 1]$$

oraz

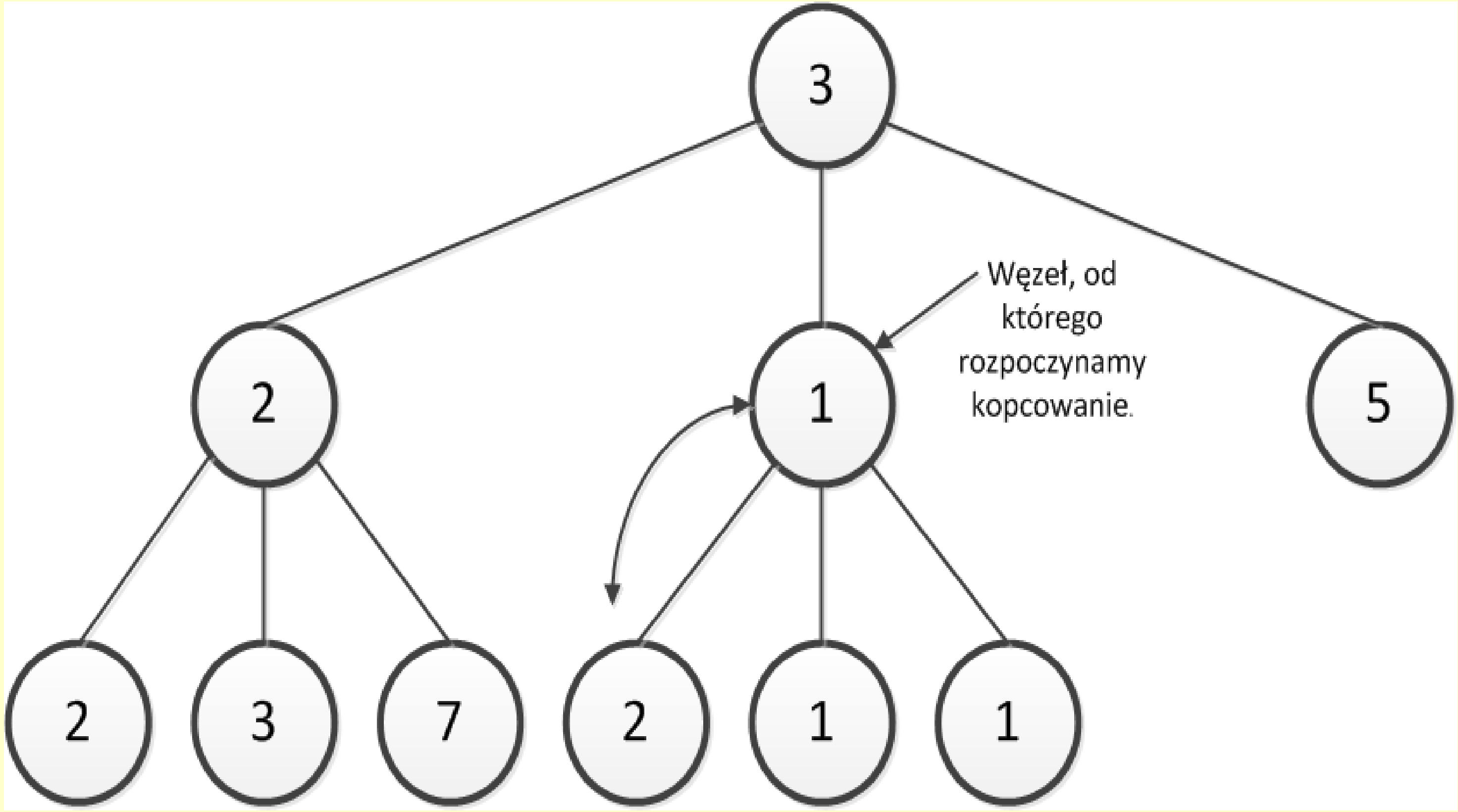
$$a[i] \geq a[3 \cdot i + 2]$$

oraz

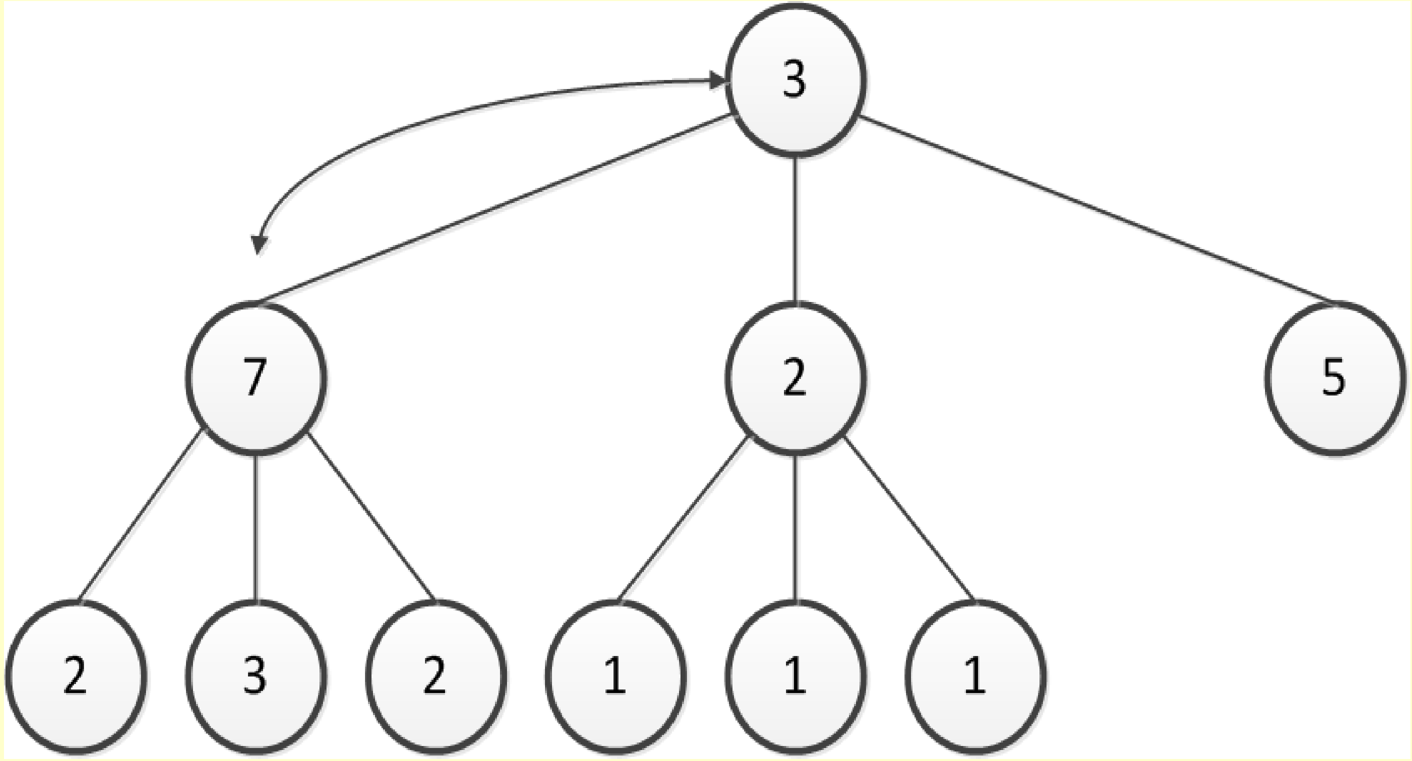
$$a[i] \geq a[3 \cdot i + 3]$$

Rozpoczynamy sortowanie od wyboru węzła

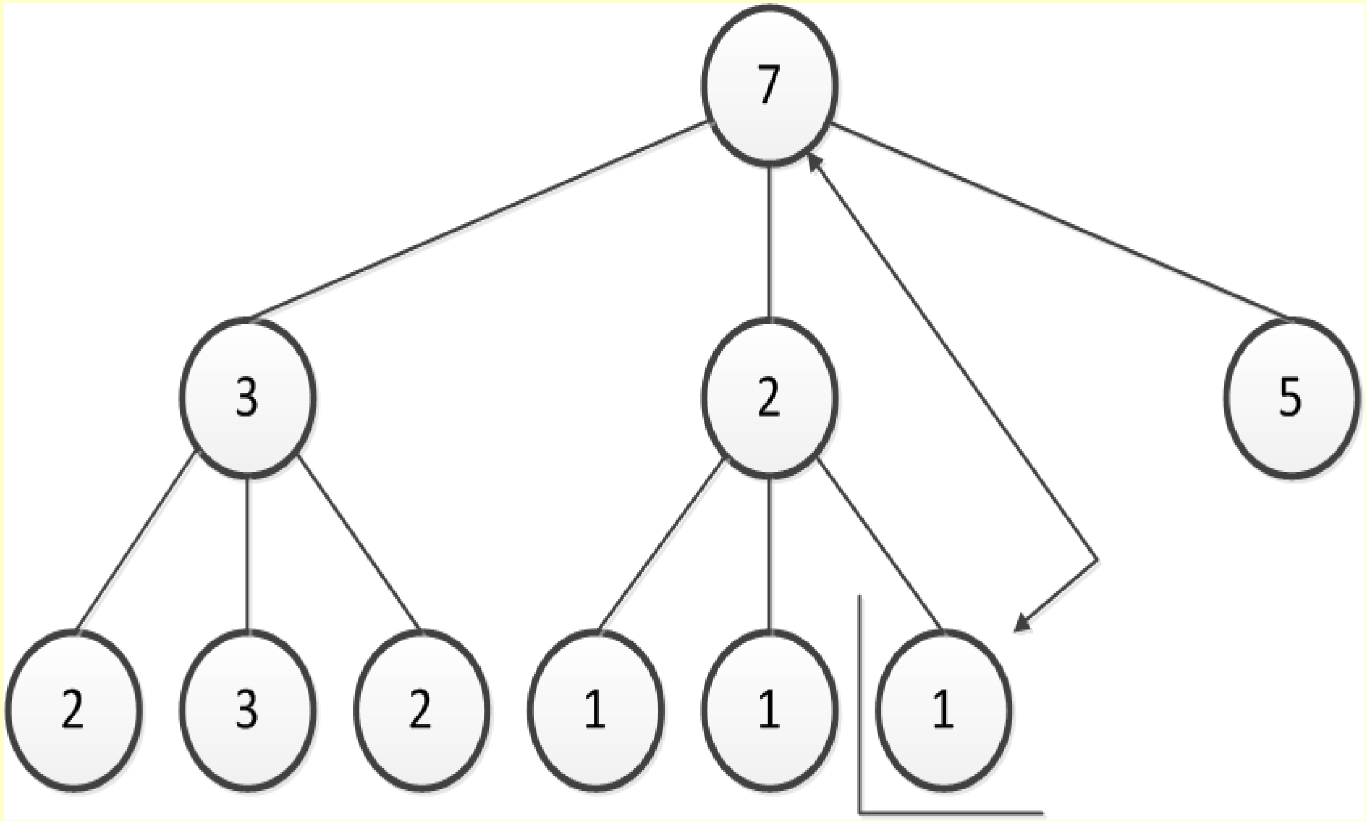
W naszym przykładzie n=10, zatem kopcowanie rozpoczynamy od węzła o numerze $\left\lfloor \frac{10-2}{3} \right\rfloor = 2$. Zaczynamy algorytm od porównywania wartości znajdującej się w węźle 2, co zostało pokazane na kolejnej ilustracji.



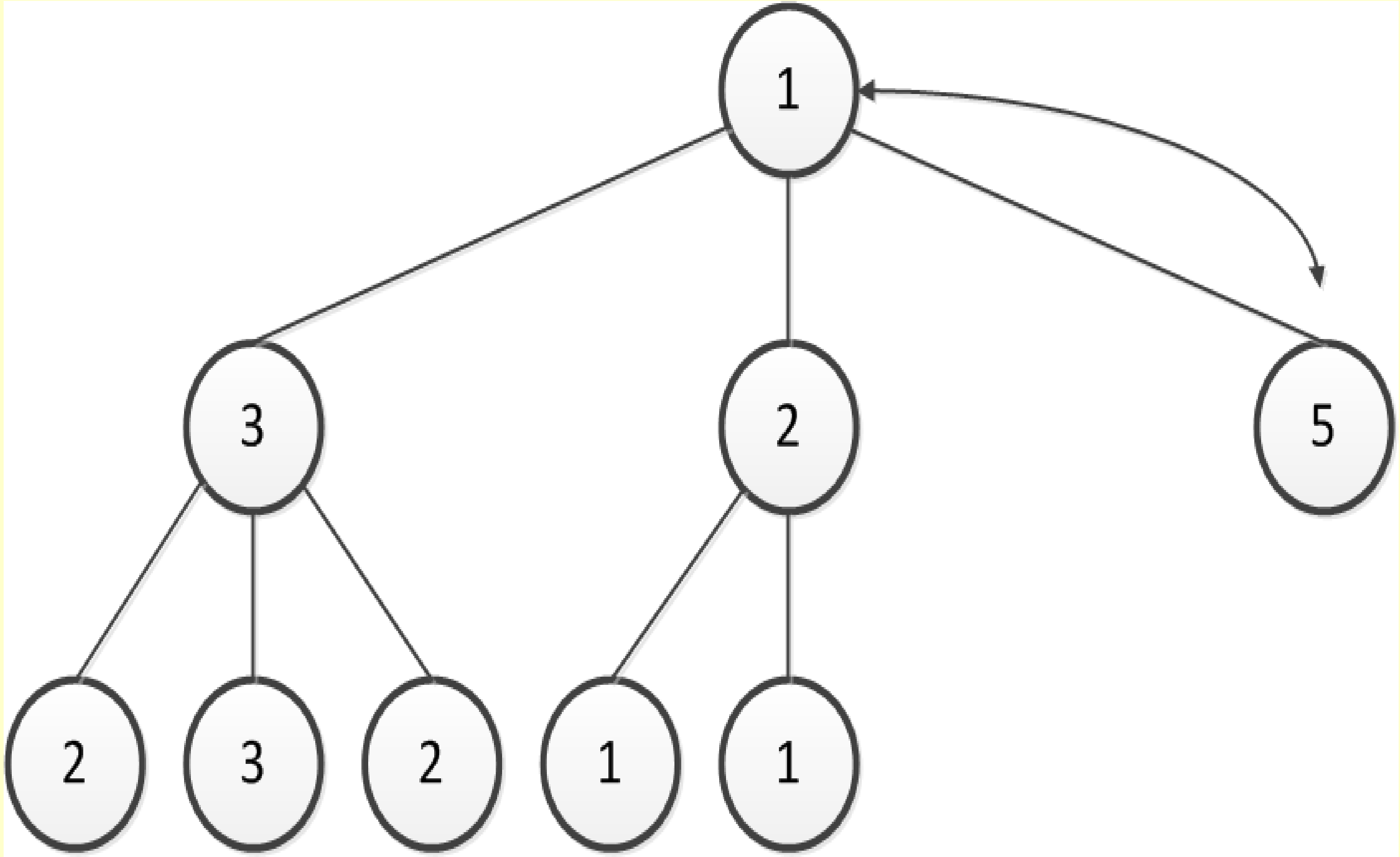
Teraz należy porównać wartości, jakie znajdują się w węzłach środkowego poziomu naszego drzewa z wartościami znajdującymi się węzłach ostatniego poziomu, które mają z nimi połączenie. Czytelnik zauważy, że wykonując algorytm postępujemy zgodnie z wzorami porządku kopca. Następuje zamiana wartości pomiędzy węzłami 2 i 7, ponieważ w węźle 7 znajduje się większa liczba, co pokazano na wcześniejszej lustracji. Kolejne porównanie wartości znajdującej się w węźle 1 z jego następnikami powoduje zamianę wartości węzła 6 i 1, czego efekt widzimy na poniższej ilustracji. Przechodząc do węzła pierwszego otrzymujemy element pokazany na poniższej ilustracji.



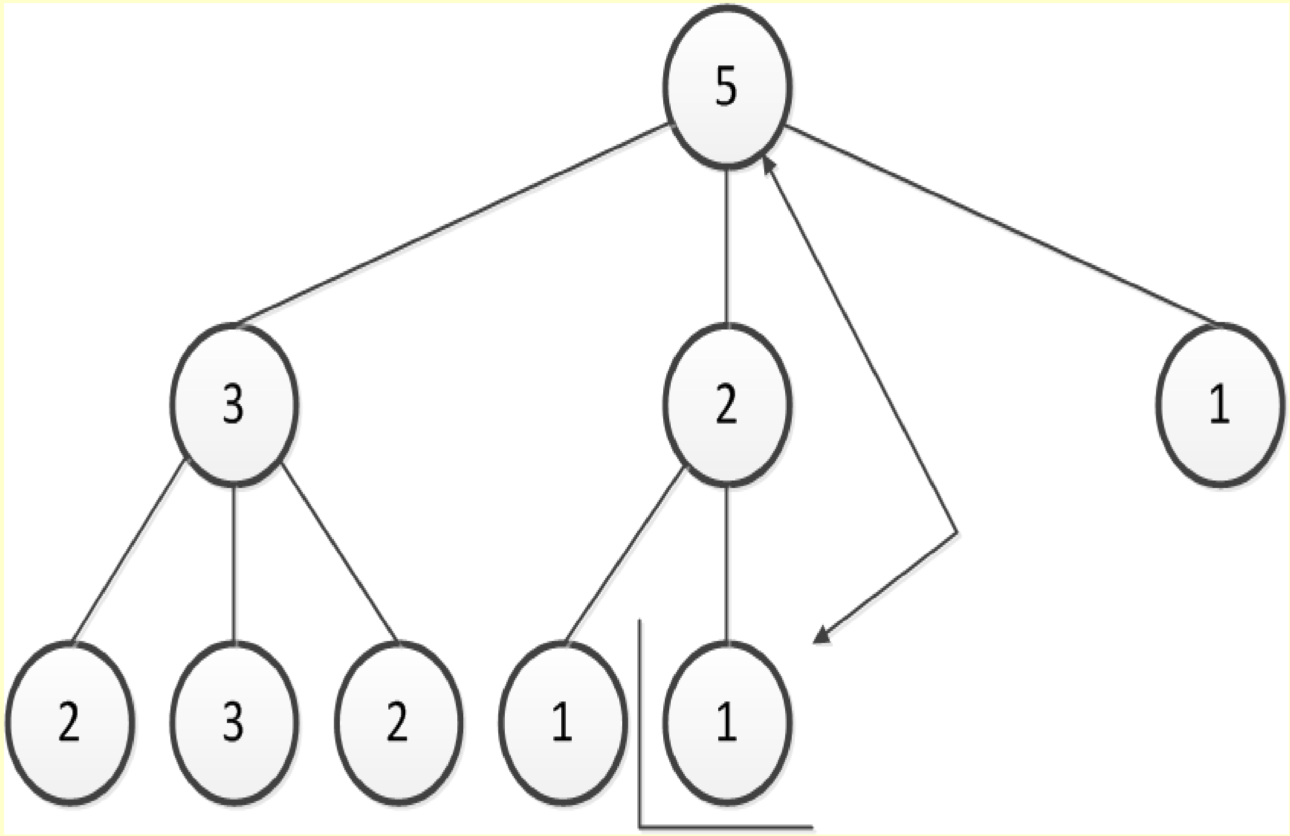
Teraz należy wykonać porównanie dla wcześniejszego węzła zgodnego z wzorami porządku kopca. Zatem przechodząc do korzenia, czyli węzła o numerze 0 po sprawdzeniu zależności wartości jego i połączonych z nim następników z poziomu kolejnego dokonujemy przesunięcia elementu największego na samą górę kopca. W efekcie otrzymujemy następujący graf 9drzewo trójdzielne).



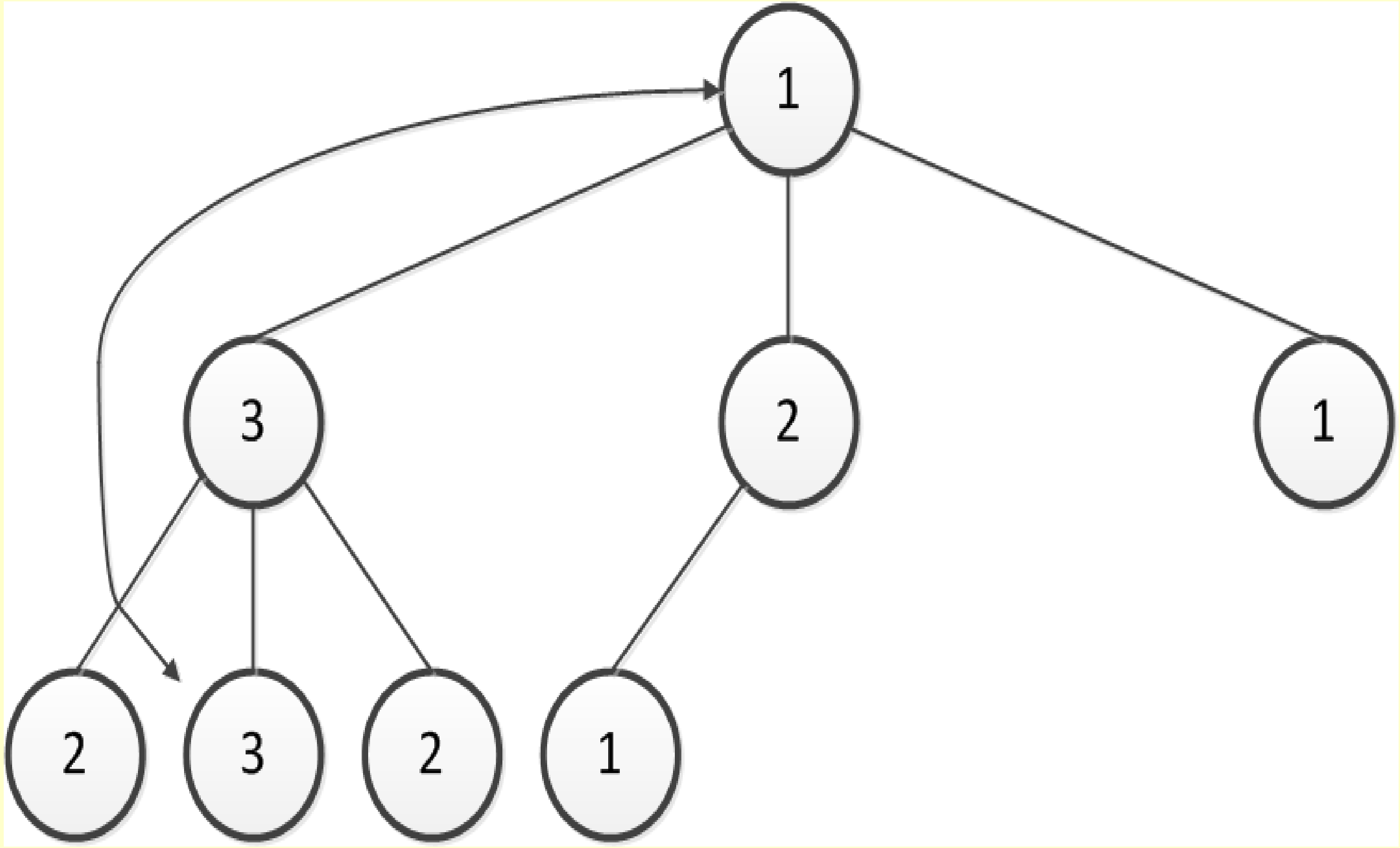
Po przestawieniach wartości w węzłach grafu dostajemy następujący ciąg liczbowy: 7,3,2,5,2,3,2,1,1,1. Następnie zamieniamy element zerowy ciągu w korzeniu z elementem ostatnim, co odpowiada ciągowi: 1,3,2,5,2,3,2,1,1,7. Następnie dokonujemy kopcowania na ciągu o jeden krótszym, ponieważ w ten sposób maksimum całego ciągu znajduje się już na samym jego końcu. Kolejne operacje kopcowania możemy przedstawić za pomocą poniższych grafów.



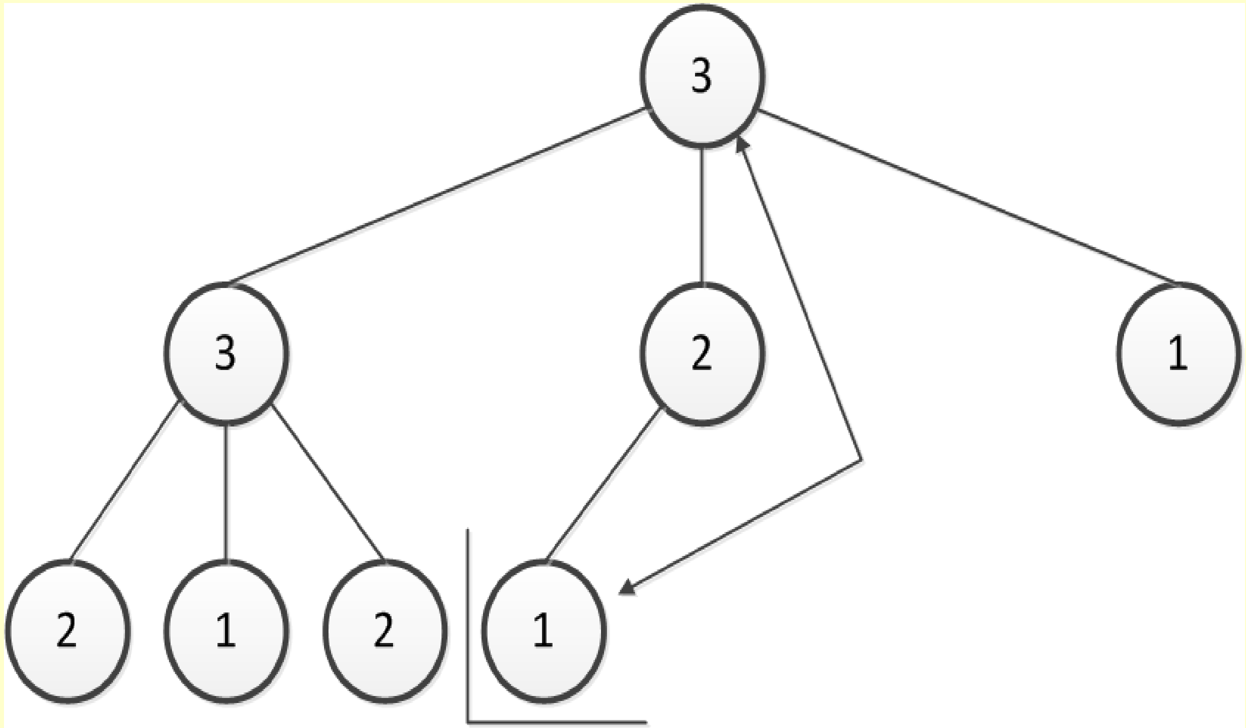
Został zamieniony element największy z poziomu środkowego z wartością znajdującą się w korzeniu. W kolejnej operacji jako największa obecnie wartość w rozpatrywanym ciągu z korzenia przechodzi ona na koniec. Element ten możemy pominąć w kolejnych operacjach, bo jest on już odpowiednio posegregowany, dlatego na kolejnej ilustracji oznaczono go specjalnym znakiem.



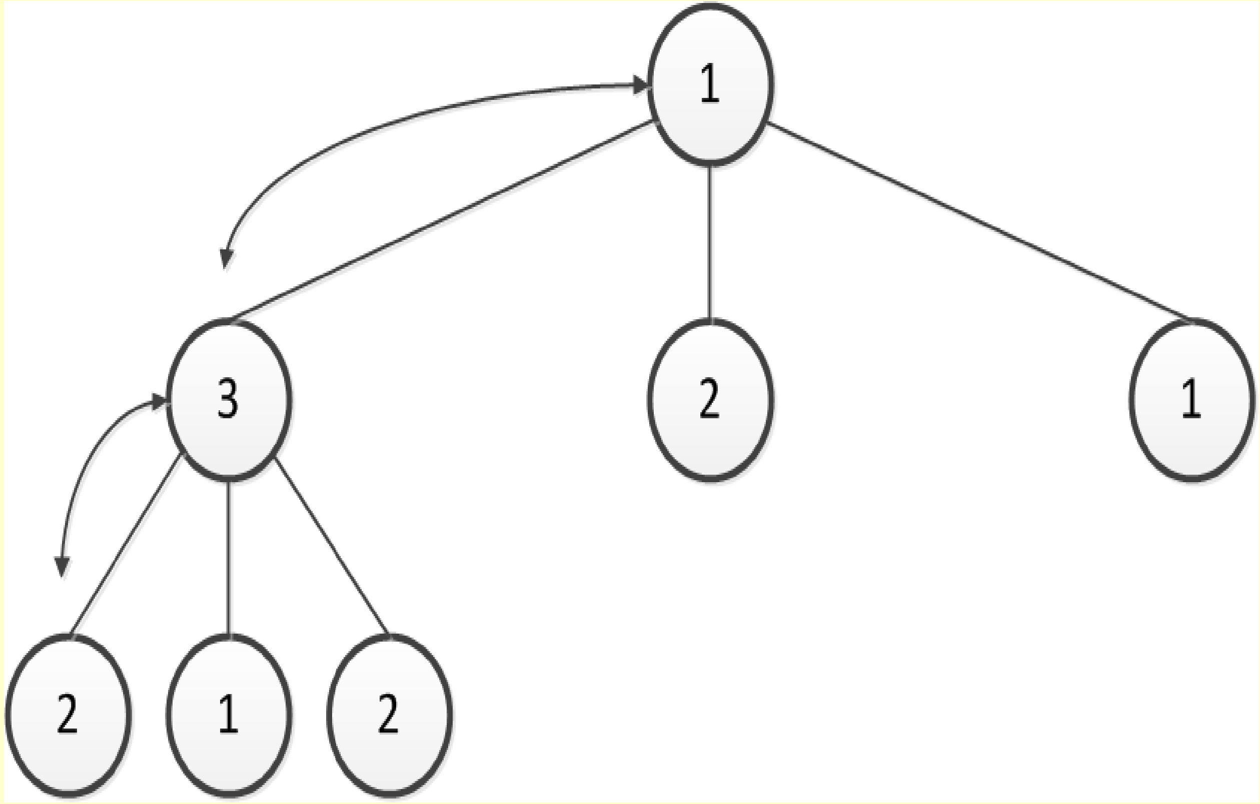
Przeprowadzona operacja daje ciąg liczbowy: 5,3,2,1,2,3,2,1,1,7 i po zmianie miejscami 1,3,2,1,2,3,2,1,5,7. W tej chwili dokonujemy kopcowania biorąc pod uwagę w przejściu ciąg o jeden element krótszy, co możemy zapisać:



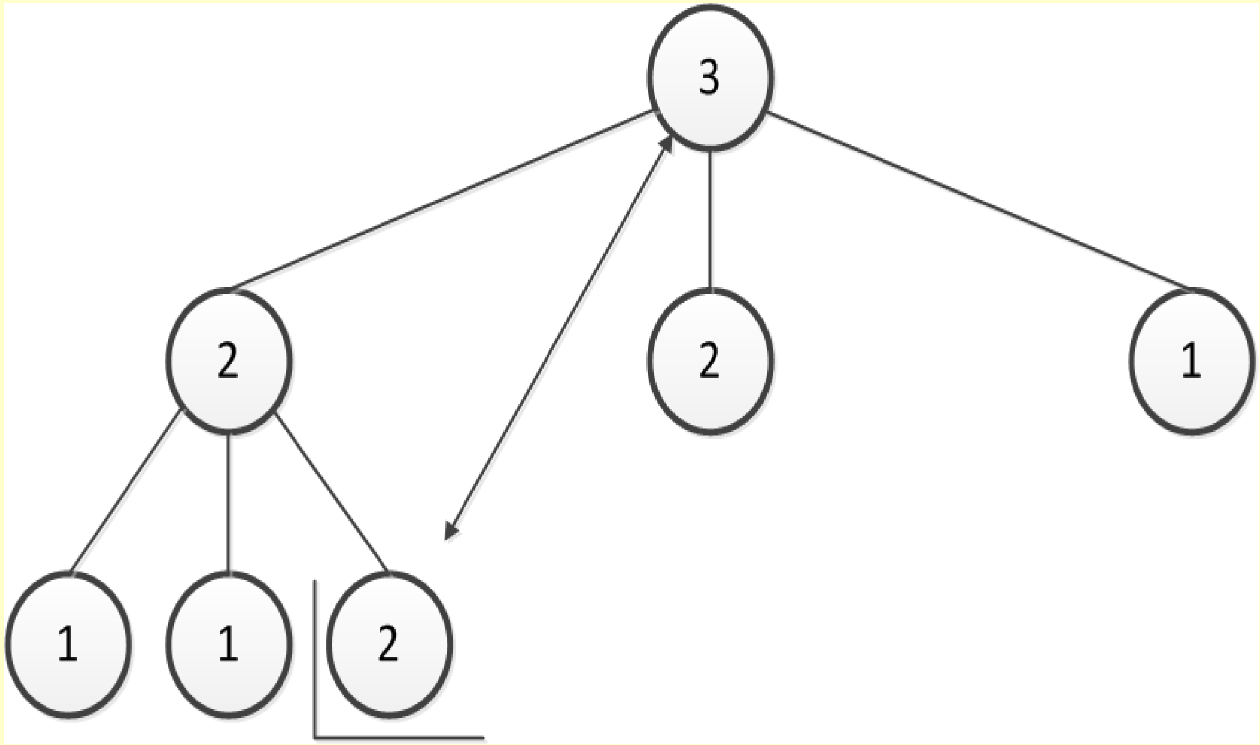
Po kolejnych porównaniach dochodzimy do ostatniego poziomu drzewa, ponieważ wszystkie węzły w poprzednich poziomach nie podlegają zamianie ze względu na wartości. Następuje zamiana wartości największej w ostatnim poziomie z elementem 0 drzewa. W ten sposób otrzymujemy graf z największym elementem w korzeniu. Dokonujemy umieszczenia tego elementu na końcu ciągu i dopisujemy go jako kolejny element posegregowany, co pokazuje kolejna ilustracja.



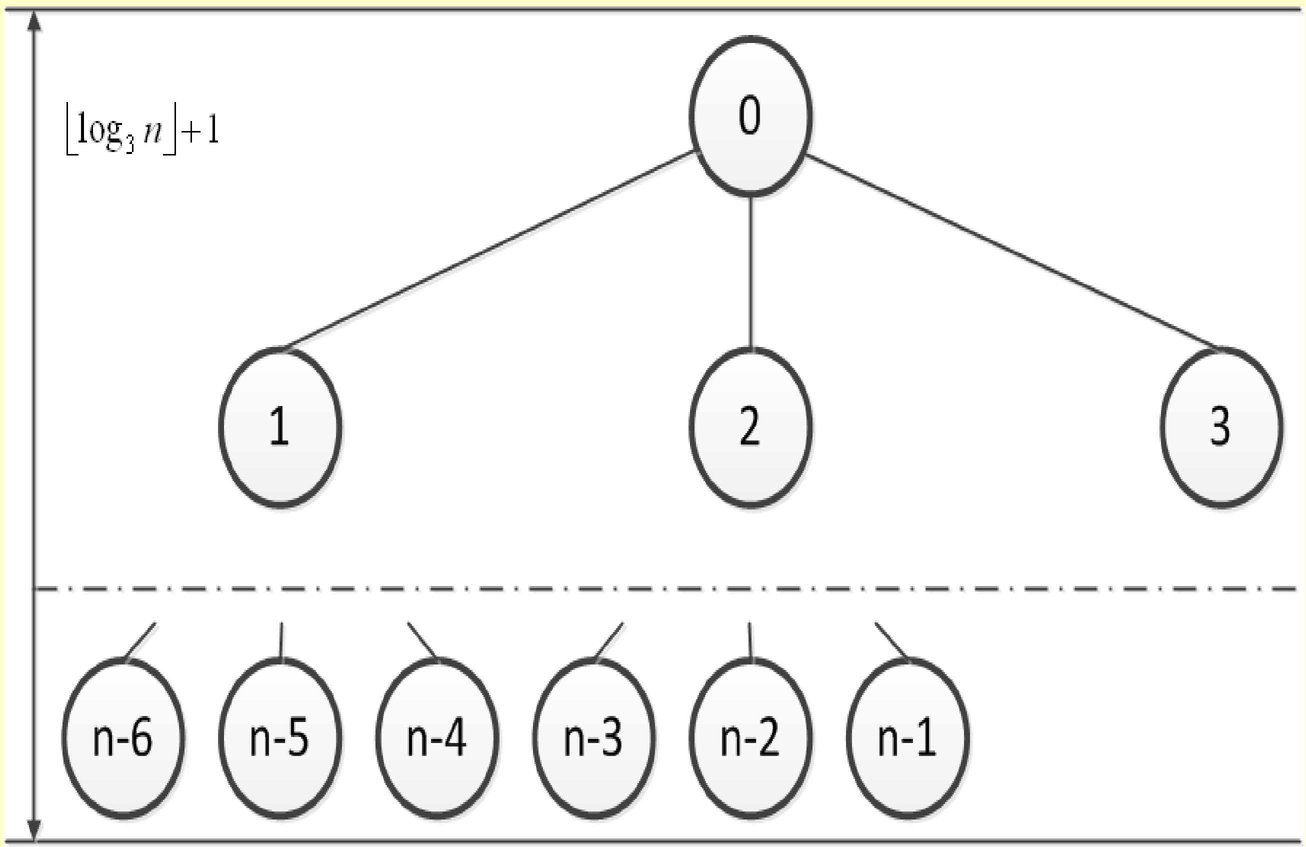
Otrzymujemy ciąg 1,3,2,1,2,1,2,1,3,5,7. Rozpoczynamy ponownie przeszukiwanie naszego drzewa w celu wyłonienia największego elementu. W tym przejściu algorytmu znajduje się ona w węźle o numerze 1. Na ilustracji zostały pokazane zamiany, jakie dokona algorytmu kopcowania trójdzielnego. Po wykonaniu operacji nasz ciąg ma postać 3,2,2,1,1,1,2,3,5,7.



Ponieważ największy element znajduje się w korzeniu zamieniamy go z ostatnim elementem rozpatrywanym ciągu i w ten sposób otrzymujemy ciąg posortowany. Po zamianie elementu zerowego z elementem szóstym ciąg ma postać 2,2,2,1,1,1,3,3,5,7. Kopcowanie przebiega dalej do uporządkowania wszystkich elementów, aby na końcu zwrócić posortowany ciąg 1,1,1,2,2,2,3,3,5,7. Zauważmy, że w kolejnych etapach rozpatrujemy odpowiednio zmniejszone kopce trójdzielne. Dzięki zastosowaniu takiej budowy kopca możemy zmniejszyć ilość wykonywanych operacji i nasz algorytm staje się efektywniejszy w porównaniu do przedstawionego poprzednio kocowania dwudzielnego.



Zauważmy, że wysokość pełnego drzewa trójdzielnego wynosi $\lfloor \log_3 n \rfloor + 1$. Wynika to bezpośrednio z definicji funkcji logarytmicznej. Symbol $\lfloor\ \rfloor$ jest rozumiany, podobnie jak poprzednio, jako zaokrąglenie $\log_3 n$ do najmniejszej liczby całkowitej w dół. Stąd czas działania algorytmu sortowania przez kopcowania wynosi $\vartheta(n \log_3 n)$. Sytuację możemy zilustrować w następujący sposób.



Zanim przeanalizujemy kod programu realizującego kopcowanie trójdzielne porównajmy metodę binarnego kopcowania z metodą kopcowania trójdzielną. Aby móc dokonać porównania tych metod należy stwierdzić, że do znalezienia największej liczby w ciągu trzyelementowym należy użyć dwóch porównań, natomiast w ciągu czteroelementowym trzech porównań. Stąd dzieląc czas działania sortowania binarnego przez sortowania trójdzielnego otrzymujemy współczynnik określający stosunek efektywności poszczególnych metod, który wynosi

$$\frac{2\log_2 n}{3\log_3 n} = \frac{\frac{2\log n}{\log 2}}{\frac{3\log n}{3}} = \frac{2\log 3}{3\log 2} \approx 1{,}0566$$

Jak widzimy sortowanie przez kopcowanie trójdzielne jest o kilka procent szybsze od sortowania binarnego, co ma duże znaczeni przy sortowaniu obszernych zbiorów danych. Ponadto zyskujemy w realizacji algorytmu na fakcie, że numeracja drzewa trójdzielnego rozpoczyna się od zera tak samo jak indeksacja tablicy w języku C/C++.

Przykład 49

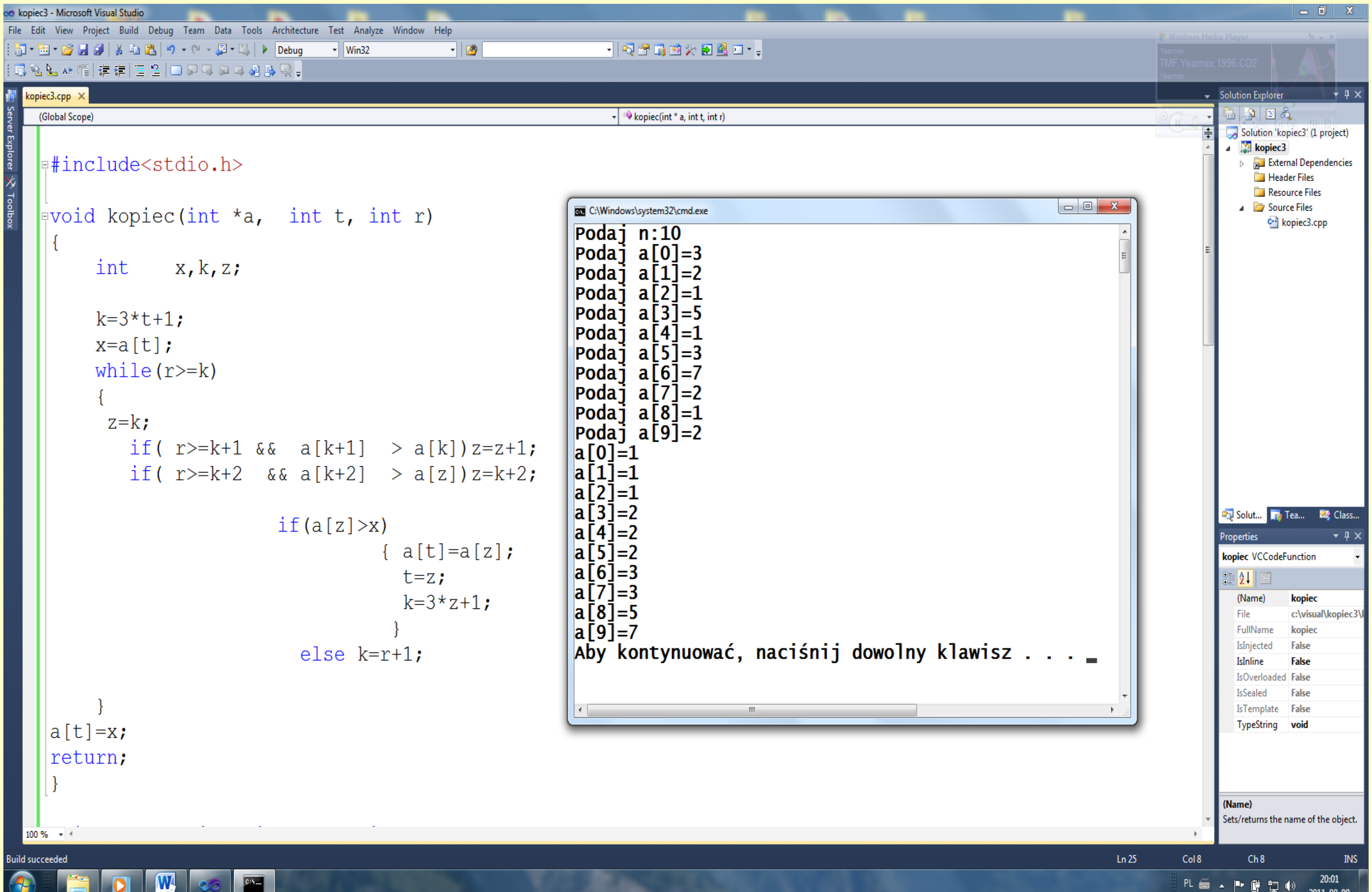
Implementacja prezentowanego algorytmu sortowania poprzez kopcowanie trójdzielne.

`#include<stdio.h>`	// Deklaracja bibliotek
`void kopiec3(int *a,  int t, int r)`	// Procedura przeszukiwania kopca
`{`	
`int    x, k, k1, k2, z;`	
`k = 3*t + 1;`	// Obliczenie indeksu lewego węzła potomnego.
`x = a[t];`	
`while(r >= k)`	
`{`	/* Przeszukuj drzewo, jeśli istnieje co najmniej jeden element potomny. */
`z = k;`	/* Przyjęcie największej wartości w lewym węźle potomnym. */
`k1 = k + 1;`	
`k2 = k + 2;`	
`if( r >= k2)`	
`{`	/* Obliczenie indeksów węzłów potomnych węzła t.*/
`if(a[k1] > a[z]) z = k1;`	
`if(a[k2] > a[z]) z = k2;`	
`}`	/* Ustalenie, który z węzłów potomnych przechowuje największą wartość. */
`else { if(r >= k1)`	
`if(a[k1] > a[k]) z = k1; }`	

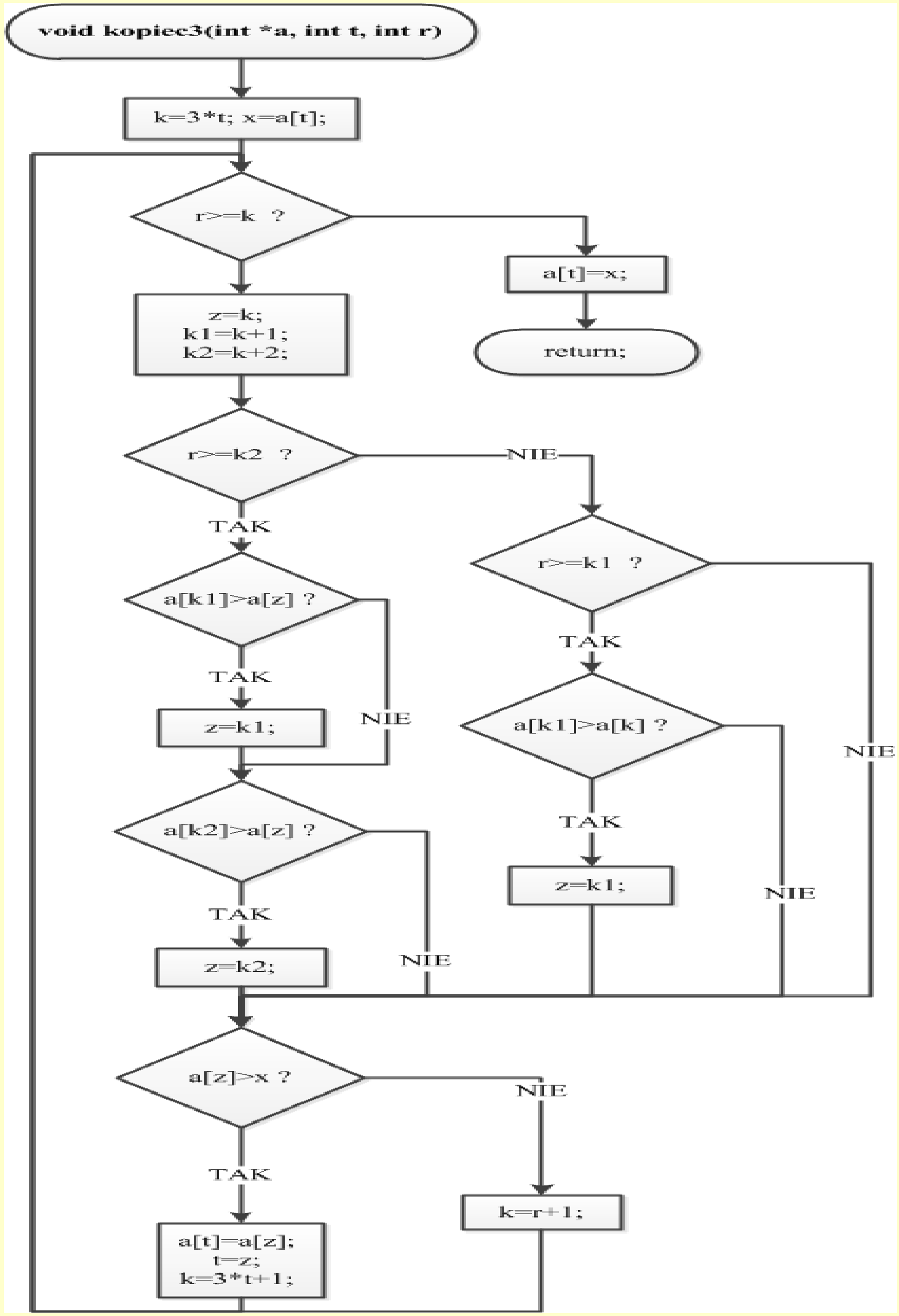
`if(a[z] > x)` `{ a[t] = a[z];` `t = z;` `k = 3*z + 1; }` `else k = r + 1;` `}` `a[t]=x;` `return;` `}`	/* Sprawdzenie, czy węzeł potomny przechowuje wartość większą od dziedzica, jeśli nie to zablokuj przeszukiwanie drzewa.*/
`void kopcowanie3( int *a, int n)` `{` `int i,x;`	// Procedura kopcowania
`for( i = ((n-2)/3); i >= 0; i--)` `kopiec3( a, i, n-1 ) ;`	// Budowa kopca.
`for(i = n-1; i>0; i--)` `{` `x = a[i];` `a[i] = a[0];` `a[0] = x;`	/* Przestawienie elementu na koniec ciągu liczbowego. */
`kopiec3(a, 0, i-1);` `}` `return;` `}`	/* Wywołanie procedury przeszukania. */

int main()	// Silnik programu
{	
int i, n;	
printf("Podaj n:");	/* Czytanie wymiaru zadania, czyli
scanf("%d", &n);	ilości węzłów w drzewie.*/
int *a= new int[n];	// Konstrukcja tablicy
for(i = 0; i < n; i++)	
{	// Wczytanie ciągu liczbowego.
printf("Podaj a[%d]=",i);	
scanf("%d", a + i);	
}	
kopcowanie3(a,n);	/* Wywołanie procedury kopcowania
	na wczytanym ciągu liczb. */
for(i = 0; i < n; i++)	
printf("a[%d]=%d\n", i, a[i]);	// Wydruk posortowanego ciągu.
}	

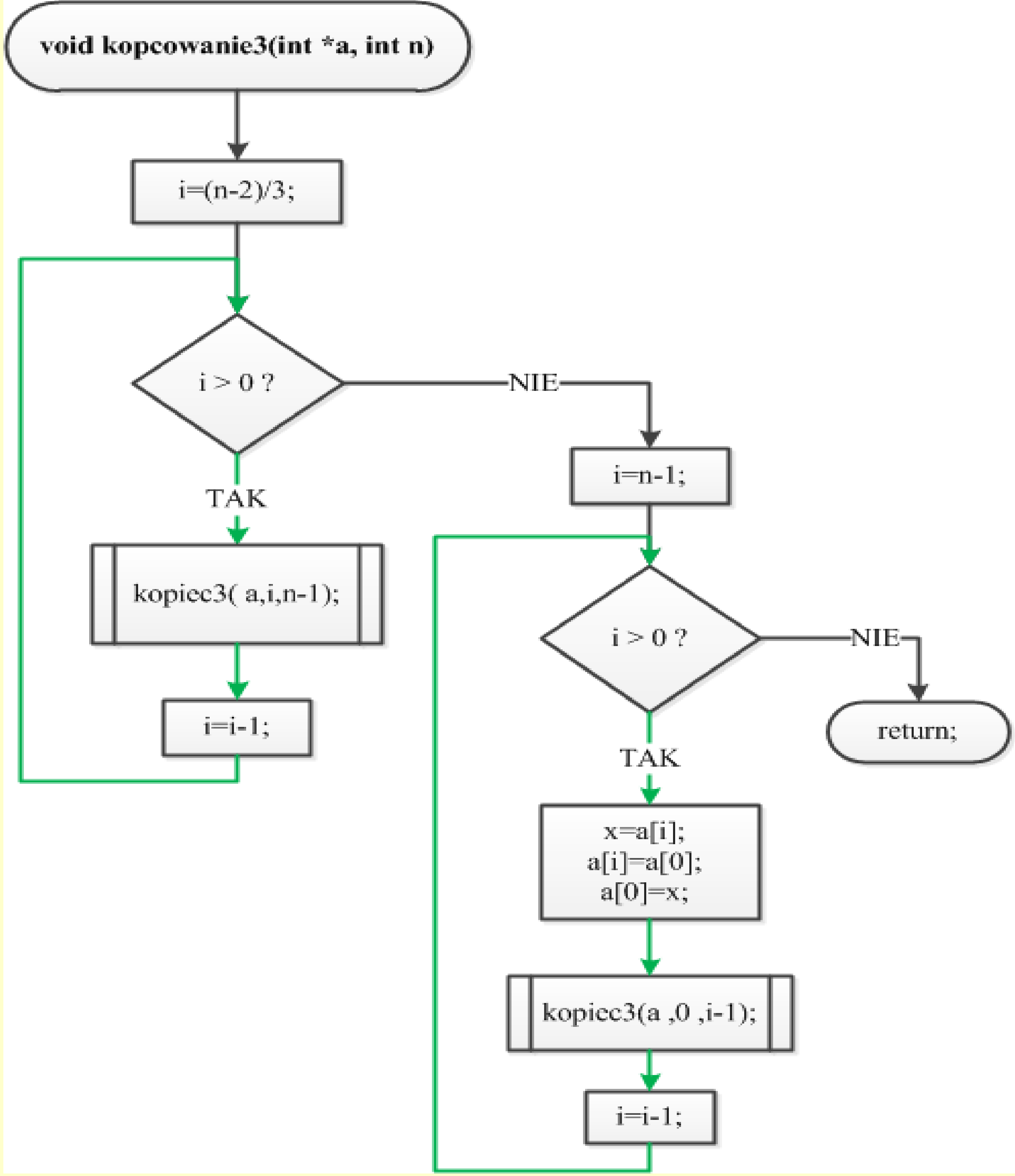
Okno kompilacji przykładu sortowania przez kopcowanie trójdzielne



- Schemat blokowy budowania kopca w sortowaniu trójdzielnym



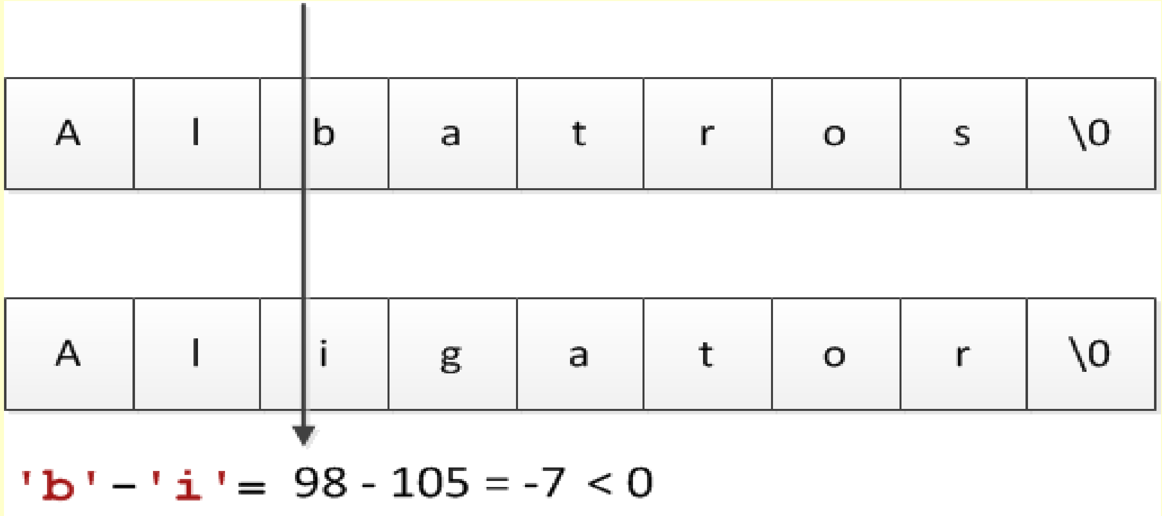
- Schemat blokowy sortowania przez kopcowanie trójdzielne.



Jak widać na powyższych schematach blokowych algorytm kopcowania trójdzielnego jest nieco trudniejszy do realizacji niż przedstawiony w poprzednim podrozdziale algorytm sortowania przez kopcowanie binarne. Ważną cechą tego algorytmu jest jego związek ze znanymi z teorii grafów i sieci grafami trójdzielnymi co zostało opisane w literaturze [11, 44-45, 66, 107]. Operując na takich strukturach w informatyce możemy dzielić dane do sortowania na odpowiednią ilość podzbiorów rozłącznych i dopiero później definiować zachodzące pomiędzy nimi relacje. Takie podejście pozwala w bardzo efektywny sposób realizować algorytmy sortowania danych. Otrzymane wyniki Czytelnik zechce porównać otrzymane po kompilacji programu wyniki z przedstawionymi na kolejnej ilustracji.

Sortowanie leksykograficzne znaków ASCII (kodów ANSI)

W poprzednich podrozdziałach przedstawiliśmy algorytmy porównywania ciągów liczbowych i określania, który z ich elementów jest większy lub mniejszy. Przedstawimy teraz metodę pozwalającą sortować dane leksykograficzne. Sortowanie leksykograficzne ma duże znaczenia dla baz danych, gdzie operujemy właśnie na wartościach leksykograficznych. W literaturze [39, 84, 114, 115] pokazano rozwinięcia i modyfikacje algorytmu sortowania leksykograficznego, które teraz omówimy. W algorytmie tym ciągi są porównywane znak po znaku, a różnica pomiędzy parą znaków decyduje o leksykograficznym porządku w ciagu. Działanie porównania prześledźmy na przykładzie. Przypuśćmy, że chcemy posortować odpowiednio dane w bazie zawierającej gatunki zwierząt. Mamy dane dwie nazwy zwierząt *albatros* i *aligator*. Oba ciągi znaków zaczynają się tą samą literą. Każdej z liter przypisujemy odpowiedni kod ANSI, zgodnie z tabelami przedstawionymi na początku podręcznika w rozdziale 3.6. Porównanie ciągów pokazano na ponższej ilustracji.



Znaki zaczynają różnić się dopiero na miejscu trzecim. Jak widać na ilustracji różnica w kodzie ANSI wynosi -7. Oznacza to, że będziemy potrzebowali do porównania funkcji mogącej operować na wszystkich wartościach całkowitych. W programie wykorzystamy taką właśnie standardową funkcję deklarowaną w następujący sposób:

```
int strcmp( const char *string1, const char *string2)
```

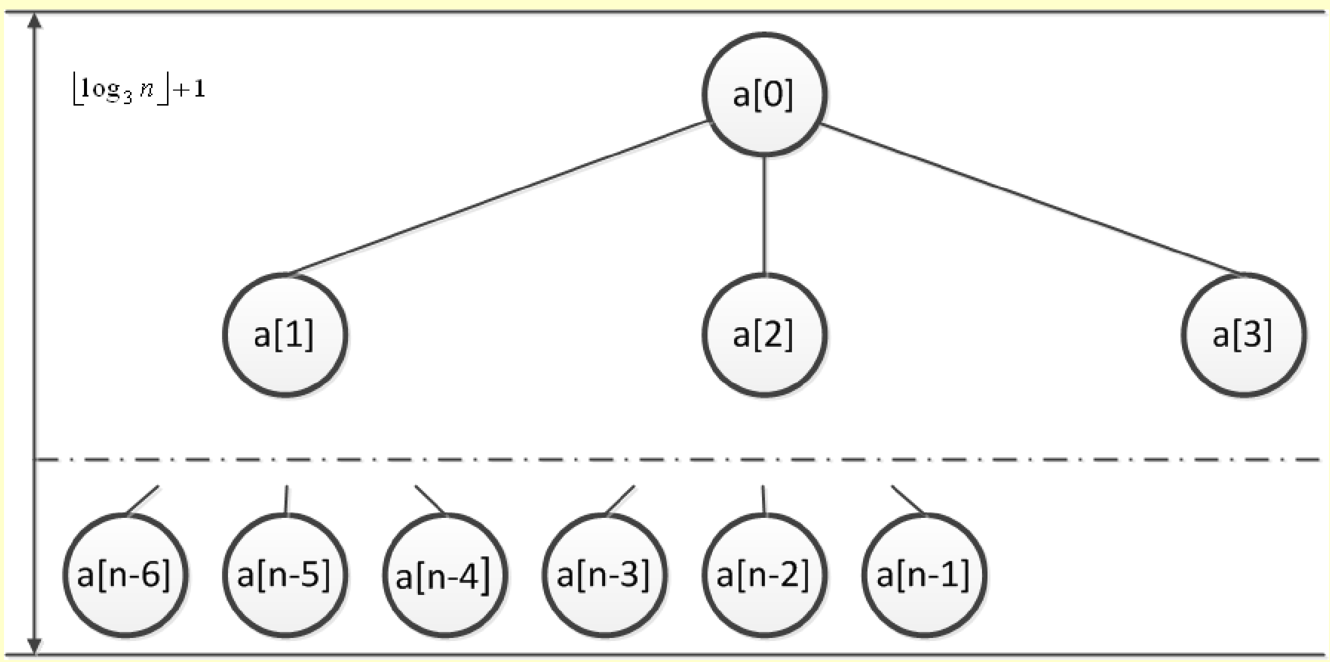
Strcmp zwraca wartość pokazaną w poniższej tabeli.

Wartość	Zależność pomiędzy ciągami
< 0	Ciąg pierwszy jest mniejszy od drugiego.
0	Ciągi są równe.
> 0	Ciąg pierwszy jest większy od drugiego.

Na podstawie wartości zwracanej przez funkcję *strcmp()* możemy przypisać odpowiednim ciągom leksykograficznym wartości numeryczne. W ten sposób sprowadzimy problem sortowania leksykograficznego do znanego już z wcześniej przedstawionych metod sortowania struktur numerycznych. Do sortowania ciągów znakowych opisanych numerycznie wykorzystamy algorytm kopcowania trójdzielnego.

Wystarczy zauważyć, że w węzłach drzewa trójdzielnego wpisane będą wskaźniki do sortowanych ciągów znakowych. Następnie porównywać będziemy ciągi znakowe za pomocą funkcji *strcmp()* aby uzyskać odpowiednio posegregowaną struktur numeryczną. Taką strukturę odkodujemy na dane leksykograficzne odwracając proces porównywania z kodem ASCII dzięki zapamiętanym przyporządkowaniem.

W omawianej metodzie można zapisać w sposób ogólny drzewo trójdzielne, w którym będą przechowywane zakodowane ciągi leksykograficzne, co pokazano na poniższej ilustracji.



Przykład v50

Napisać program wczytujący linie tekstu zapisanego w zbiorze słownik.txt i wypisujący posortowane linie leksykograficznie na monitor. Ilość linii jest ograniczona do 1023, natomiast ilość znaków w linii jest ograniczona do 1023

`#include<stdio.h>` `#include<stdlib.h>` `#include<string.h>` `#include<conio.h>` `#define N 1024` `#define M 1024`	/* Deklaracja bibliotek i stałych programu.*/

```c
void kopiec3(char **a,  int t, int r)
{
    char    *x;
    int k, k1, k2, z;
    k = 3*t + 1;
    x = a[t];
    while(r >= k)
    {
        z = k;
        k1 = k + 1;
        k2 = k + 2;
        if( r >= k2)
        { if(strcmp(a[k1], a[z]) > 0) z = k1;
          if(strcmp(a[k2], a[z]) > 0) z = k2;  }
         else
        { if(r >= k1) if(strcmp(a[k1], a[k]) > 0) z = k1;}
         if(strcmp(a[z], x) > 0)
        {
            a[t] = a[z];
            t = z;
            k = 3*z + 1;
         }
         else k=r+1;
      }
     a[t]=x;
     return;
}
```

/* Omówiony we wcześniejszym rozdziale algorytm budowy kopca. */

<pre>void kopcowanie3(char **a, int n) { int i; char *x; for(i = ((n-2)/3); i >= 0; i--) kopiec3(a, i, n-1) ; for(i = n-1; i > 0; i--) { x = a[i]; a[i] = a[0]; a[0] = x; kopiec3(a, 0, i-1); } return; }</pre>	/* Omówiony we wcześniejszym rozdziale algorytm sortowania trójdzielnego. */

```c
int main()
{
   char **a;
   int n=0, i;
   FILE *f;
   a = (char **)malloc(sizeof(char*)*N);
   a[0] = (char *)malloc(sizeof(char)*M);

   if((f=fopen("slownik.txt", "r")) == NULL)
   {
         printf("Nie moge otworzyc pliku slownik.txt\n");
         _getch();
          return 0;
    }
    while(fgets(a[n], 1023, f) != NULL)
    {
        n++;
        a[n]=(char *)malloc(sizeof(char)*M);
     }
     kopcowanie3(a,n);

      for(i = 0; i < n; i++)
          fputs( a[i], stdout );
      fclose(f);
     _getch();
      return 0;
}
```

/* Utworzenie wektora wskaźników dla kolejnych wierszy, maksymalnie 1023 wiersze plus jeden na zakończenie wpisywania */

/* Rezerwacja miejsca na wprowadzoną linię tekstu, maksymalnie 1023 znaki plus jeden na znak '\0'. */

/* Otwarcie pliku zawierającego wiersze do posortowania. */

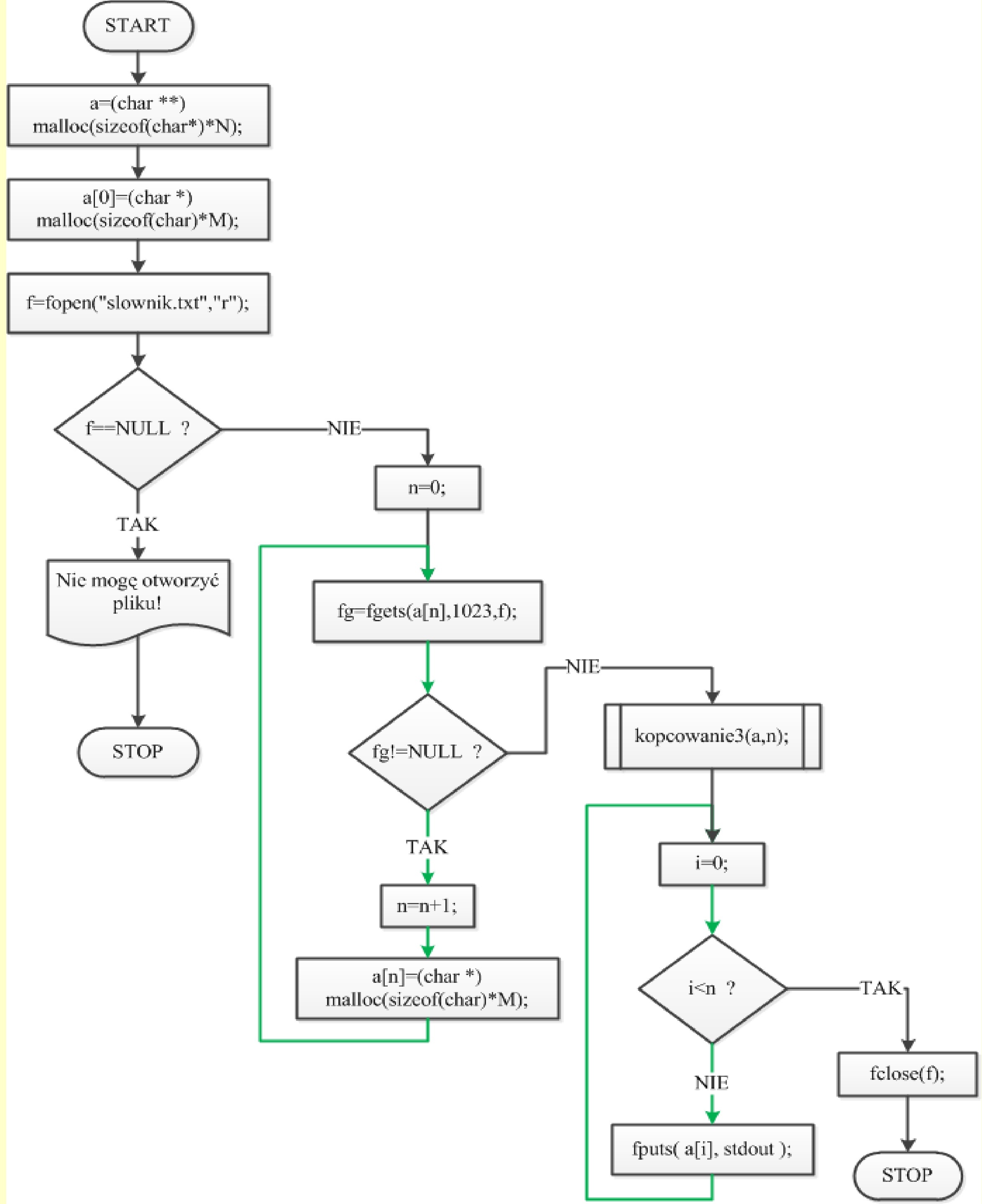
/* Wczytanie kolejnych wierszy tekstu. Po wczytaniu wiersza rezerwowane jest miejsce na kolejny wiersz. */

/* Sortowanie przez kopcowanie trójdzielne na wskaźnikach do tekstu. */

/* Wydruk posortowanych linii tekstu. */

- Schemat blokowy sortowania leksykograficznego

Schemat blokowy procedury sortowania został pokazany na wcześniejszej ilustracji. W procedurze została wykorzystana stworzona dla metody kopcowania trójdzielnego procedura kopcowania, co zostało zaznaczona poniższym schemacie odpowiednim elementem.



Do posortowania wybraliśmy nazwy miast niezawierające liter polskich. Zagadnienie sortowania znaków narodowych w innych systemach kodowania przekracza zakres niniejszej książki i odsyłamy czytelnika do literatury [39, 84, 114, 115]. Przykładowy tekst do posortowania pobrany z bazy danych może wyglądać następująco.