# Lekcja 6: Sterty i kolejki priorytetowe

## Wstęp

Prawie cała poprzednia lekcja była poświęcona drzewom binarnym służącym do sortowania i wyszukiwania danych. Wszystkie one spełniały warunek, że lewy potomek jest zawsze mniejszy od ojca, a ten z kolei jest mniejszy od prawego potomka. Wystarczyło we właściwy sposób przejść przez takie drzewo, by uzyskać elementy posortowane. Cały problem tkwił w tym, jak takie drzewo zbudować.

W tej lekcji pokażemy drzewa, w których ojciec jest zawsze większy (lub co najmniej równy) obu swoim potomkom. Taka struktura pozwoli na natychmiastowe wyjęcie z drzewa elementu największego - o najwyższym priorytecie. Cały problem, jak to zrobić, by ten największy zawsze był w korzeniu - gotów do pobrania. I o tym jest właśnie ta lekcja...

## Kolejki priorytetowe

Jednym z podstawowych typów struktur danych są **kolejki priorytetowe**. Przechowują one elementy pewnego zbioru danych (oznaczanego najczęściej jako S), na którym jest określona relacja porządku – jest ona realizowana poprzez przydzielenie kluczy (priorytetów) elementom zbioru. Z takiej kolejki elementy są wyjmowane nie w kolejności ich pojawienia się w kolejce, lecz zależnie od priorytetów. Kolejki priorytetowe dzieli się na dwie grupy:

- typu **"max"**
- typu **"min"**

W zależności od potrzeby (gdy chcemy wyjmować z kolejki najpierw elementy o największym lub też najmniejszym priorytecie) stosuje się kolejki jednego z dwóch wymienionych typów (odpowiednio typu max lub min).

Kolejka priorytetowa musi być zaimplementowana w taki sposób, aby można było wykonywać na niej niżej wymienione operacje:

- **Insert(S, x)** – wstawienie elementu x do kolejki S
- **FindMax(S)** – poszukiwanie w kolejce S elementu o największym kluczu; funkcja zwraca ten element jako wynik
- **DelMax(S)** – usunięcie z kolejki S elementu o największym kluczu; funkcja zwraca ten element jako wynik

Powyższe funkcje są wykonywane dla kolejki typu "max" i w dalszej części rozdziału skoncentrujemy się na kolejkach tego rodzaju (algorytmy dla kolejek typu "min" są „bliźniacze").

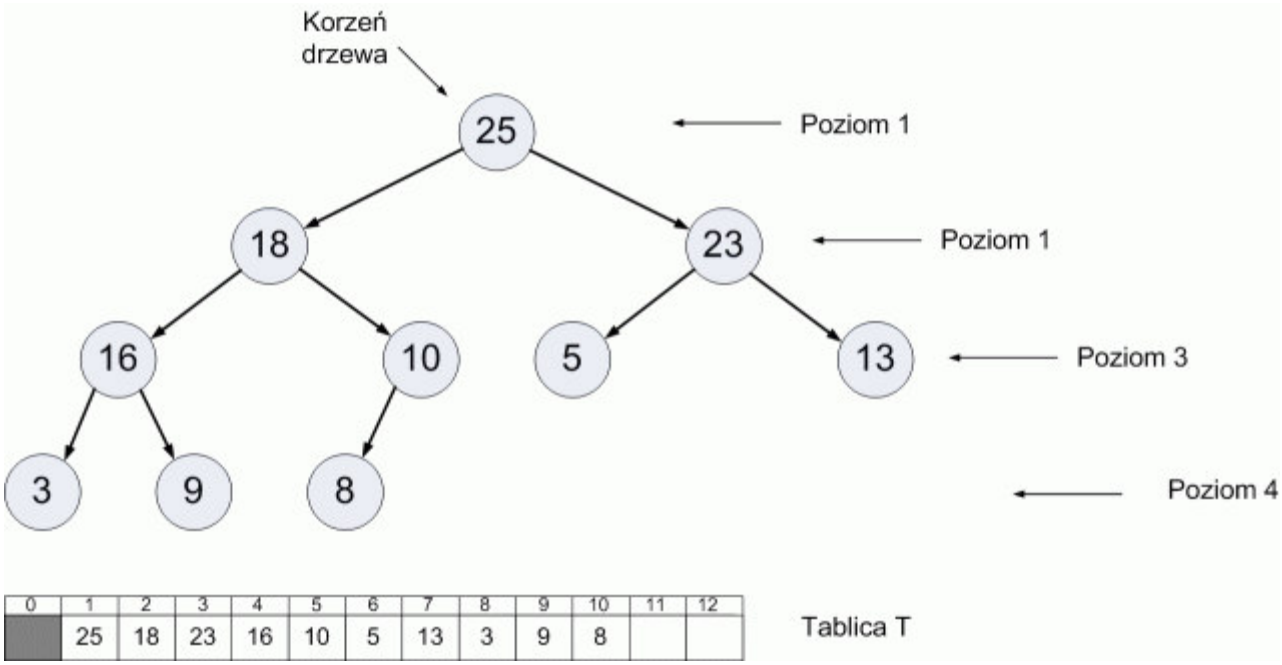
Zbiór podstawowych funkcji operujących na kolejkach priorytetowych często rozszerza się o dodatkowe metody, takie jak usuwanie z kolejki dowolnego elementu czy zmiana wartości klucza na większy (*IncreaseKey*).

Kolejki priorytetowe można implementować na kilka różnych sposobów. Najbardziej oczywistym i zarazem najprostszym sposobem jej utworzenia jest wykorzystanie klasycznej listy. Nie daje ona jednak dużej efektywności przy wykonywaniu wymienionych wyżej operacji. Znacznie częściej stosowana jest struktura **kopca** (ang. *heap*) – w literaturze polskiej drugą spotykaną nazwą (będącą odpowiednikiem angielskiego *heap*) jest **sterta**.

### Kopiec czyli sterta

Po zrozumieniu zasady tworzenia i funkcjonowania kopca, będziemy już wiedzieli, dlaczego ta właśnie struktura jest wyjątkowo efektywna jako implementacja kolejki priorytetowej. Otóż kopiec jest najczęściej zapisywany przy użyciu tablicy jednowymiarowej, ale zapisane w nim elementy należy sobie wyobrażać jako węzły drzewa binarnego, i tak właśnie są one obrazowane.

Typowy wygląd kopca przedstawia rysunek:




Każdy poziom drzewa jest całkowicie wypełniony (co określa się mianem drzewa pełnego na danym poziomie) – liczba elementów na k-tym poziomie jest równa $2^k$. Jedynie ostatni, najniższy poziom drzewa może nie być zapełniony do końca, a wszystkie liście znajdujące się na nim występują od lewej strony. Kopiec zapisuje się w tablicy jednowymiarowej (tu oznaczonej jako tablica T) w taki sposób, że korzeń drzewa jest przechowywany w elemencie o indeksie 1, z kolei jego elementy będące lewym i prawym potomkiem są zapisywane w komórkach o indeksach odpowiednio 2 i 3; następny poziom drzewa będzie miał przydzielone indeksy 4-7 i tak dalej. Dzięki takiej koncepcji składowania danych w tablicy jest możliwe uzyskanie indeksu ojca oraz obu potomków danego węzła w sposób natychmiastowy.

Mianowicie, jeśli interesujący nas węzeł drzewa posiada indeks i, to:

- indeks ojca wynosi $\lfloor i/2 \rfloor$ - czyli część całkowita z połowy wartości i
- indeks lewego potomka wynosi $2i$
- indeks prawego potomka wynosi $2i+1$

Wygodnie jest zapisać te relacje między indeksami w postaci funkcji, które często będą wykorzystywane:

Funkcja **indeksLewegoSyna** (**i** całkowite) typu całkowitego

```
1.  return 2*i;
```

Funkcja **indeksPrawegoSyna** (**i** całkowite) typu całkowitego

```
1.  return 2*i+1;
```

Funkcja **indeksOjca** (**i** całkowite) typu całkowitego

```
1.  return i/2;     // UWAGA: dzielenie całkowite
2.                  // ponieważ zwracamy zmienną typu całkowitego, więc automatycznie
3.                  // część ułamkowa wyniku i/2 zostanie „obcięta”
```

Z faktu, że należy wartość zmiennej `i` należy pomnożyć bądź podzielić przez 2, wynika prostota tych operacji w kodzie maszynowym - wystarczy proste przesunięcie bitów w lewo lub w prawo.

Jednak najważniejszą cechą sterty, której istnienie w sposób znaczący zachęca do użycia struktury kopca jako kolejki priorytetowej, jest tzw. **własność kopca**, którą dla danych zapisanych w tablicy T można sformułować następująco:

## $T[i] \leq T[indeksOjca(i)]$

co oznacza, że **każdy element w kopcu ma wartość nie większą niż jego ojciec** (sprawdźcie to na rysunku powyżej).

Oczywiście podana własność nie dotyczy przypadku i=1. Korzeń drzewa nie posiada przecież ojca, sam znajduje się na najwyższym poziomie.

Jeżeli powyższa własność kopca jest stale utrzymywana, to **największy element** spośród całego zbioru danych zapisanych w drzewie znajduje się **zawsze na pozycji korzenia**. Zatem funkcja wyszukania największego elementu kolejki priorytetowej zapisanej w postaci sterty polega na pobraniu go z korzenia, czyli z pierwszego elementu tablicy T, a więc wynosi dokładnie **O(1)**. Cały problem polega na utrzymaniu własności kopca po usunięciu elementu z drzewa (najczęściej korzenia) lub wstawieniu nowego rekordu do kolejki priorytetowej.

Tablica T ma dodatkowo dwa atrybuty: rozmiar tablicy oraz rozmiar sterty, określający, ile elementów tablicy T (począwszy od T[1]) należy do struktury kopca. Nic nie stoi na przeszkodzie, by komórki T[rozmiar_sterty+1]..T[rozmiar_tablicy-1] były używane jako pamięć pomocnicza - w dalszej części rozdziału dowiemy się, że bywają one używane do zapisywania elementów pobranych ze sterty (algorytm sortowania przez kopcowanie). Aby powiązać zmienną rozmiar_sterty z tablicą T, najwygodniej jest utworzyć nowy **typ rekordowy** o nazwie **Tsterta** i dwu polach:

- **tablica T** o n elementach
- **rozmiar_sterty**

Możemy skorzystać z nowo powstałego typu rekordowego i utworzyć zmienną `S` typu `Tsterta`, przechowującą kopiec.

Dla uproszczenia przyjmijmy, że w miejscach, gdzie będziemy używali pojęcia tablicy T, odnosić się to będzie do pola T rekordu `S` typu `Tsterta`.

Aby pokazać Wam działanie poszczególnych metod operujących na stercie, przedstawimy je w dalszej części lekcji w sposób opisowy oraz graficzny. Czytając opis zawarty w niniejszym rozdziale, warto zaglądać do przykładu rysunkowego, znajdującego się w rozdziale następnym.

## Operacje wykonywane na stercie

Zacznijmy od podstawowej funkcji, której zadanie polega na przywróceniu własności kopca – wywoływana jest ona wtedy, gdy zachodzi podejrzenie, że prawidłowa struktura kopca mogła zostać naruszona. Najczęściej taka sytuacja może nastąpić, gdy z kopca zostanie usunięty korzeń – czyli pobrany zostanie największy element kolejki priorytetowej.

Sama metoda przywracania własności kopca polega na sprawdzeniu, czy dane poddrzewo (powiedzmy, o korzeniu o indeksie i-tym) spełnia cechy sterty – tzn. czy element pod indeksem i-tym jest nie mniejszy niż dwaj jego synowie. Jeśli tak nie jest, to należy zamienić ojca z większym z synów (czyli zachodzi „spłynięcie w dół" elementu), a następnie należy po raz kolejny sprawdzić, czy struktura kopca jest spełniona – tym razem dla poddrzewa o indeksie korzenia, gdzie znajduje się element, który przed chwilą przesunął się z poziomu o 1 wyższego. Aby cała sterta posiadała własność kopca, tę własność musi mieć każde jej poddrzewo.

Dla sprecyzowania wiedzy o tej metodzie przedstawiamy poniżej jej pseudokod:

Procedura **przywrocWlasnoscKopca** (**S** typu Tsterta, **i** całkowite)

```
1.  lewy=indeksLewegoSyna(i)
2.  prawy=indeksPrawegoSyna(i)
3.  if (lewy <= S.rozmiar_sterty oraz S.T[i] < S.T[lewy])
4.      max=lewy
5.  else
6.      max=i
7.    if (prawy<=S.rozmiar_sterty oraz S.T[max] < S.T[prawy])
8.      max=prawy
9.  if (max nie jest równy i) {
10.     zamień miejscami elementy S.T[i] oraz S.T[max]
11.     i=max
12.     przywrocWlasnoscKopca(S, i)     // rekurencyjne wywołanie
13. }
```

Tak jak wcześniej wspomnieliśmy, gdy następuje zamiana miejscami ojca z jednym z jego synów, trzeba wywołać funkcję przywracania własności kopca dla poddrzewa, którego korzeń „spłynął" z poziomu wyższego. Najprostszym w zapisie sposobem implementacji jest tu użycie rekurencji – będziemy wywoływać funkcję `przywrocWlasnoscKopca` do momentu, aż zostaną wykonane wszystkie niezbędne zamiany ojców z potomkami.

Skoro poznaliśmy powyższą procedurę, to możemy się zastanowić, jak zbudować stertę posiadając dane zebrane w tablicy, o których nic wiemy. Otóż przy użyciu metody `przywrocWlasnoscKopca` jesteśmy w stanie utworzyć z tych danych strukturę sterty w bardzo prosty sposób. Elementy tablicy T tworzą drzewo, które (z bardzo dużym prawdopodobieństwem) nie jest stertą. Należy więc każde z jego poddrzew zmodyfikować w taki sposób, by posiadało strukturę sterty. Każde drzewo jednoelementowe jest kopcem, więc metodę `przywrocWlasnoscKopca` wywołujemy dla każdego poddrzewa T zawierającego więcej niż 1

węzeł – od najmniejszych poddrzew aż dla poddrzewa będącego całym drzewem:

procedura **BudujKopiec** (**S** typu Tsterta)

```
1. // n - rozmiar tablicy S.T
2. S.rozmiar_sterty = n
3. for (i=n/2; i>=1; zmniejszaj i o 1)  // dzielenie całkowite n/2
4.     przywrocWlasnoscKopca(S, i)
```

Skoro już wiemy, jak przywracać własność kopca (oraz jak tworzyć kopiec z losowych danych), możemy w tym momencie przedstawić sposób wykonywania jednej z głównych funkcji działających na kolejkach priorytetowych. Mianowicie, w celu pobrania największego elementu kolejki, usuwamy korzeń ze struktury sterty, następnie przesuwamy najbardziej prawy liść z najniższego poziomu drzewa (czyli de facto element tablicy T o największym indeksie należący do sterty) na miejsce usuniętego korzenia, a następnie musimy, jeśli to konieczne, przywrócić własność kopca, zaczynając od korzenia całego drzewa.

Ogólną strukturę powyższej metody można pokazać następująco:

Funkcja **pobierzNajwiekszy** (**S** typu Tsterta) typu takiego jak T

```
1. największy = S.T[1]
2. S.T[1]= S.T[rozmiar_sterty]   // przesuń „ostatni” liść na miejsce pobranego korzenia
3. zmniejsz S.rozmiar_sterty o 1
4. przywrocWlasnoscKopca(S, 1)
5. return  największy
```

Drugą metodą niezbędną do prawidłowego funkcjonowania kolejki priorytetowej jest procedura dodania nowego elementu – oznaczona na początku rozdziału jako Insert(S, x). Idea działania tej procedury jest przejrzysta:

- najpierw dodajemy wstawiany element jako nowy liść na najniższym poziomie
- a następnie, w razie potrzeby, przesuwamy ten element na wyższe poziomy (zamieniając z kolejnymi ojcami), aż do momentu, gdy całe drzewo będzie posiadało własność kopca

Procedura **wstawElement** (**S** typu Tsterta, **x** typu takiego jak T)

```
1. zwiększ S.rozmiar_sterty o 1
2. i = S.rozmiar_sterty
3. S.T[i]=x
4. while (i>1 oraz x>S.T[indeksOjca(i)] {
5.     zamień S.T[i] z elementem S.T[indeksOjca(i)]
6.     i=indeksOjca(i)
7. }
```

Ta wersja jest najbardziej klasyczna i odpowiada wersji rysunkowej w następnym rozdziale. Procedurę tę można też spotkać w wersji trochę "na skróty", gdzie zamiast zamiany elementów stosuje się przesuwanie elementów, które miałyby pełnić rolę ojca dodanego elementu, na coraz niższe poziomy, jak tylko okazuje się, że one się do tej roli nie nadają. Gdy już odpowiedni ojciec (odpowiednio duży) we właściwym miejscu drzewa się znajdzie, dostawiany element wpisywany jest od razu jako jego syn na zwolnione w poprzednim kroku miejsce; w szczególnym przypadku miejsce to może okazać się korzeniem.

Procedura **wstawElement_m** (**S** typu Tsterta, **x** typu takiego jak T)

```
1. zwiększ S.rozmiar_sterty o 1
2. i = S.rozmiar_sterty
3. while (i>1 oraz x>S.T[indeksOjca(i)] {
4.     S.T[i]=S.T[indeksOjca(i)]
5.     i=indeksOjca(i)
6. }
7. S.T[i]=x
```

W tym momencie wiemy już, jak funkcjonuje kolejka priorytetowa zrealizowana za pomocą kopca – czyli jak funkcjonują metody dodawania do kolejki nowego elementu oraz usuwania z niej największego. Warto jednak zobrazować ich działanie w sposób rysunkowy, co pokażemy w następnym rozdziale. Teraz zaś zobaczycie, jak nową strukturę danych można wykorzystać do sortowania.

## Sortowanie kopcowe

W lekcji 3 podręcznika, omawiającej nieelementarne metody sortowania, takie jak *QuickSort* czy *MergeSort*, wspomnieliśmy o algorytmie sortowania przez kopcowanie. Należy on, podobnie jak dwa wyżej wspomniane, do grupy algorytmów o złożoności obliczeniowej O(n lgn).

Przed chwilą poznaliście procedury pobierające największy element kopca. Algorytm *HeapSort* (czyli sortowania kopcowego) opiera się w głównej mierze właśnie na wykorzystaniu funkcji *PobierzNajwiekszy*, dzięki czemu reszta jego kodu jest wyjątkowo czytelna.

procedura **HeapSort** (**S** typu Tsterta)

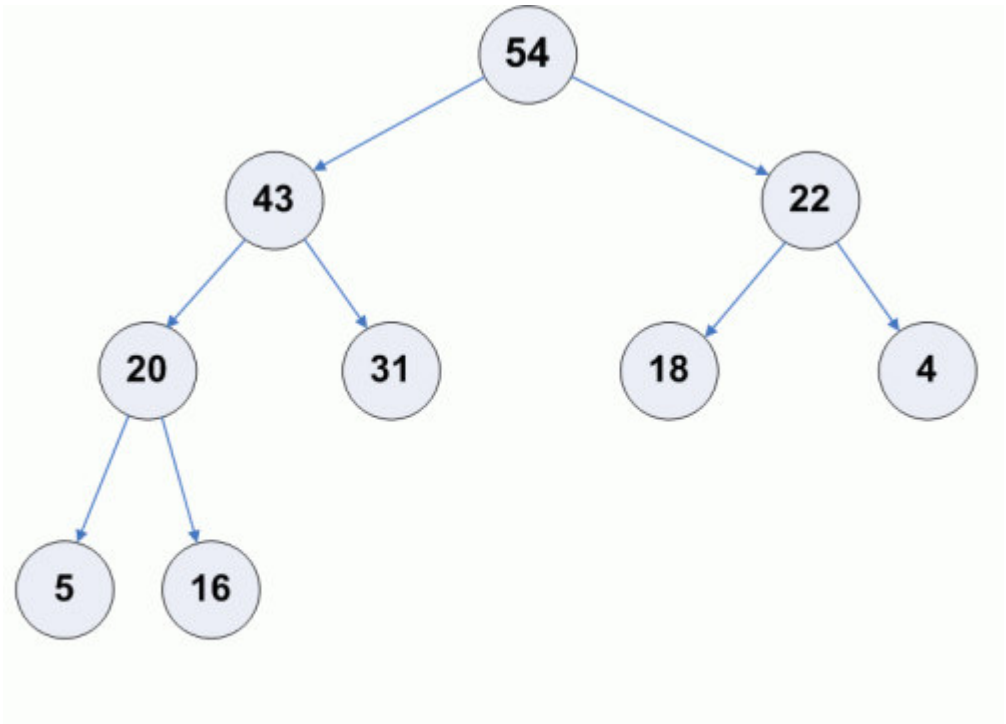
```
1. BudujKopiec(S)
2. // n - rozmiar tablicy S.T
3. for (i=n-1; i>=2; zmniejszaj i o 1){
4.     zamień S.T[1] z S.T[i]
5.     zmniejsz S.rozmiar_sterty o 1
6.     przywrocWlasnoscKopca(S,1)
7. }
```

Procedura sortowania kopcowego najpierw buduje kopiec z losowych danych zapisanych w tablicy T, następnie bierze największy element kopca (korzeń) i zamienia go z ostatnim węzłem kopca (o największym indeksie). W tym momencie pozostaje już tylko zmniejszyć rozmiar sterty i przywrócić jej prawidłową strukturę. Pobieranie kolejnych największych wartości z kopca oraz przywracanie własności kopca wykonuje się w pętli do momentu, gdy zostanie nam kopiec 2-elementowy – następuje ostatnia zamiana T[1] z T[2], po której wszystkie dane tablicy T będą juz posortowane.

Sortowanie przez kopcowanie jest użyteczną funkcją, gdyż łączy kilka istotnych zalet: czas jej wykonania jest rzędu O(n lgn), a także należy ono do metod **działających w miejscu**: oznacza to, że algorytm oprócz danych wejściowych potrzebuje tylko dodatkowej pamięci o stałym rozmiarze – co jest dużą zaletą w przypadku przetwarzania ogromnej liczby danych. Ta druga cecha sortowania kopcowego daje mu w tym względzie przewagę nad algorytmami *QuickSort* oraz *MergeSort* - które nie „działają w miejscu".

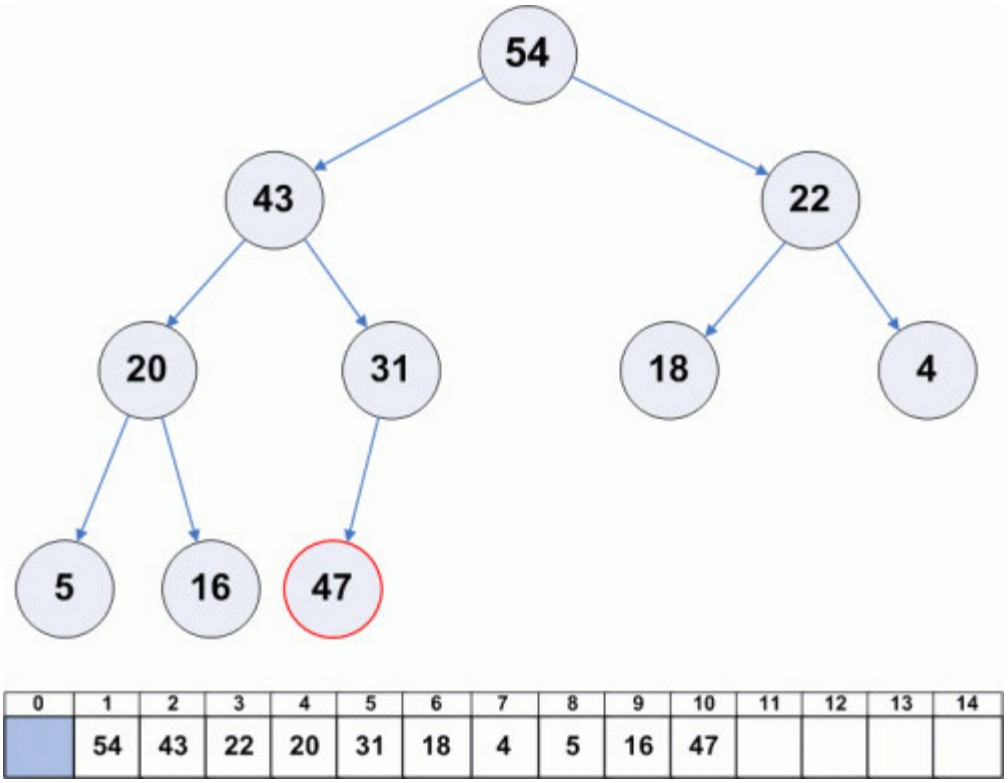
## Przykład rysunkowy

Załóżmy, że dysponujemy kopcem przedstawionym poniżej:



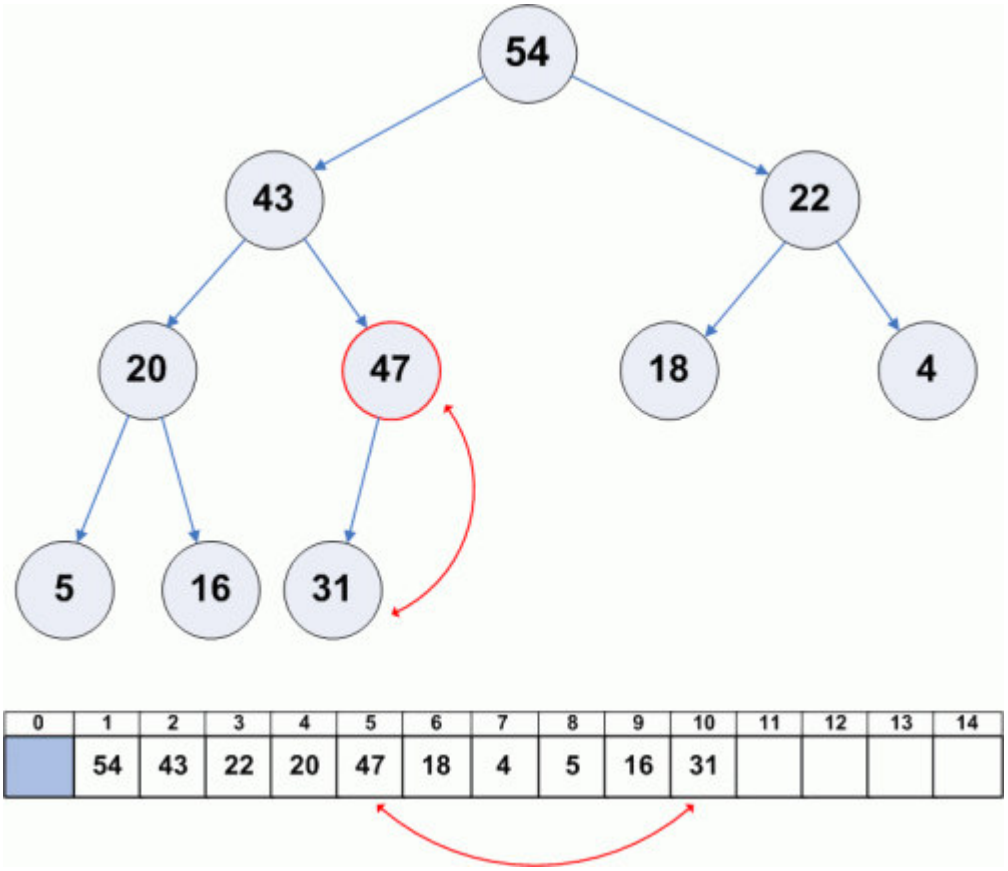

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 54 | 43 | 22 | 20 | 31 | 18 | 4 | 5 | 16 | | | | | |

Zawartość komórki T[0] jest pusta, elementy kopca są zapisywane począwszy od T[1]. Dodajemy nowy element o wartości 47 – wywołanie funkcji *wstawElement(T,47)*. Nowe elementy są zawsze dodawane na koniec tablicy:



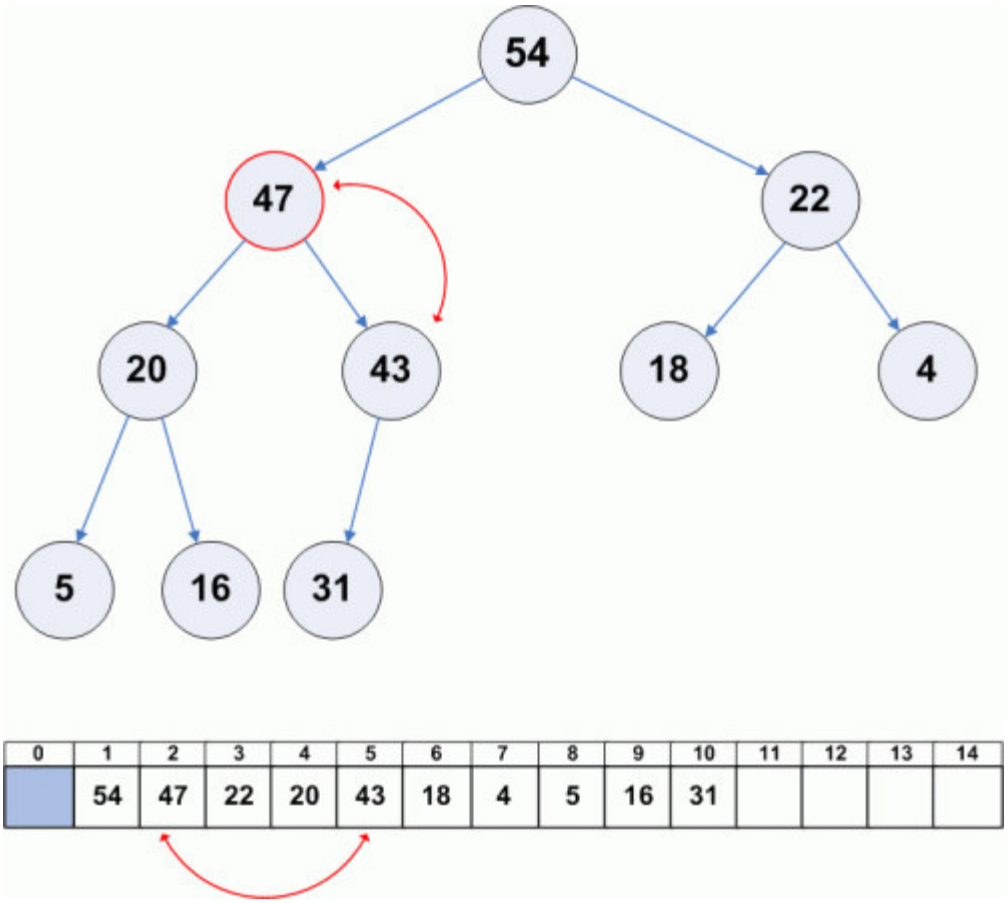

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 54 | 43 | 22 | 20 | 31 | 18 | 4 | 5 | 16 | 47 | | | | |

W tym momencie należy przywrócić strukturę sterty. Element 47 wędruje do góry aż do momentu, gdy jego ojciec nie jest mniejszy od niego. Poniżej zostały przedstawione kroki przywracania struktury sterty:



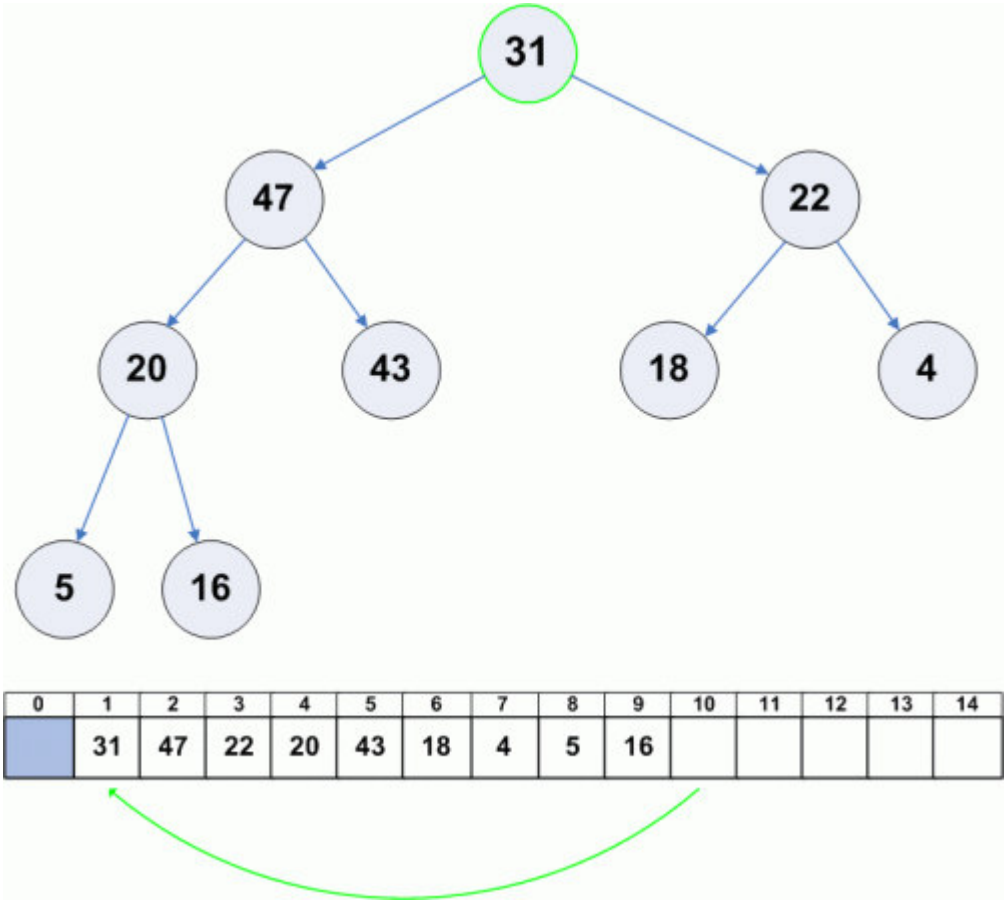

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 54 | 43 | 22 | 20 | 47 | 18 | 4 | 5 | 16 | 31 | | | | |

Element 47 zamienił się ze swym ojcem o wartości 31.



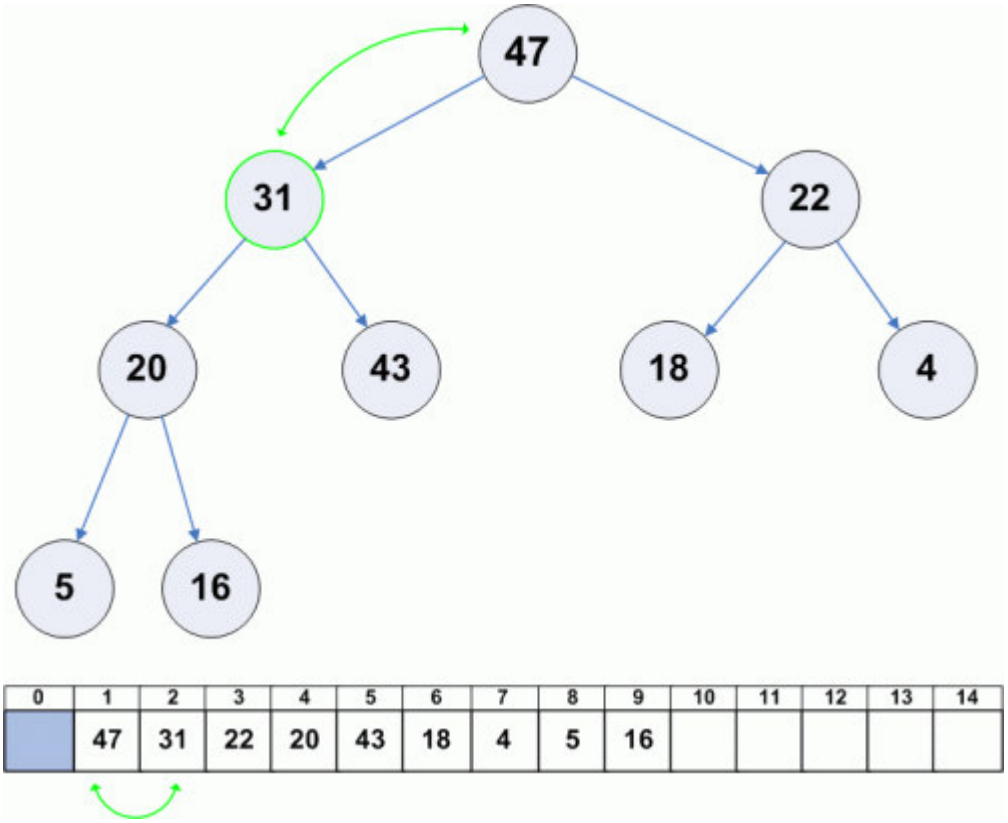

Na powyższym schemacie element 47 zamienił się ze swym nowym ojcem o wartości 43. Po dokonaniu tej wymiany, drzewu została przywrócona własność kopca. Na tym kończymy proces wstawiania elementu do sterty.

Teraz chcemy pobrać z kopca element największy – następuje wywołanie metody **pobierzNajwiekszy(T)**. Bierzemy więc korzeń drzewa (element o wartości 54), a na jego miejsce przesuwamy ostatni zapisany element w tablicy (czyli u nas jest to 31). Sterta po tej operacji wygląda następująco:

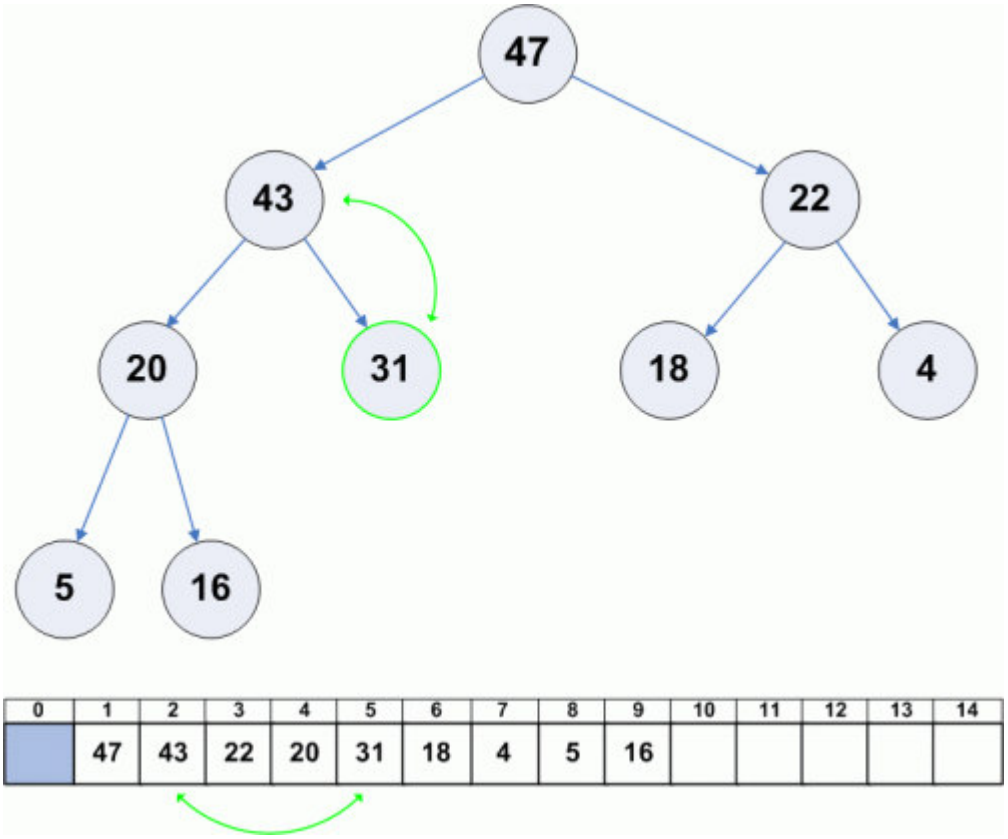



Jak widzimy, drzewo nie spełnia własności kopca, więc należy ją przywrócić. Element 31 będzie wędrował tak długo w dół, aż wszystkie warunki poprawności sterty będą spełnione.

Zaczynamy od sprawdzenia potomków węzła T[1]. Spośród dwóch synów węzła o wartości 31 lewy syn (47) ma większą wartość niż jego ojciec i "brat", więc to on "zamienia się miejscami" ze swoim ojcem. Po tej operacji sterta ma postać:




Sytuacja powtarza się. Węzeł 31 ma mniejszą wartość od jednego ze swoim synów. Tym razem jest nim węzeł o wartości 43 (prawy syn), który zamienia się ze swoim ojcem. Po tej zamianie struktura kopca wygląda tak, jak poniżej:




Jak widzimy, węzeł o wartości 31 w tym momencie nie ma potomków - a zatem nie ma już możliwości „wędrowania w dół". W rezultacie algorytm pobierania największego elementu sterty kończy swoje działanie, i własność kopca zostaje przywrócona.

## Kopce dwumianowe

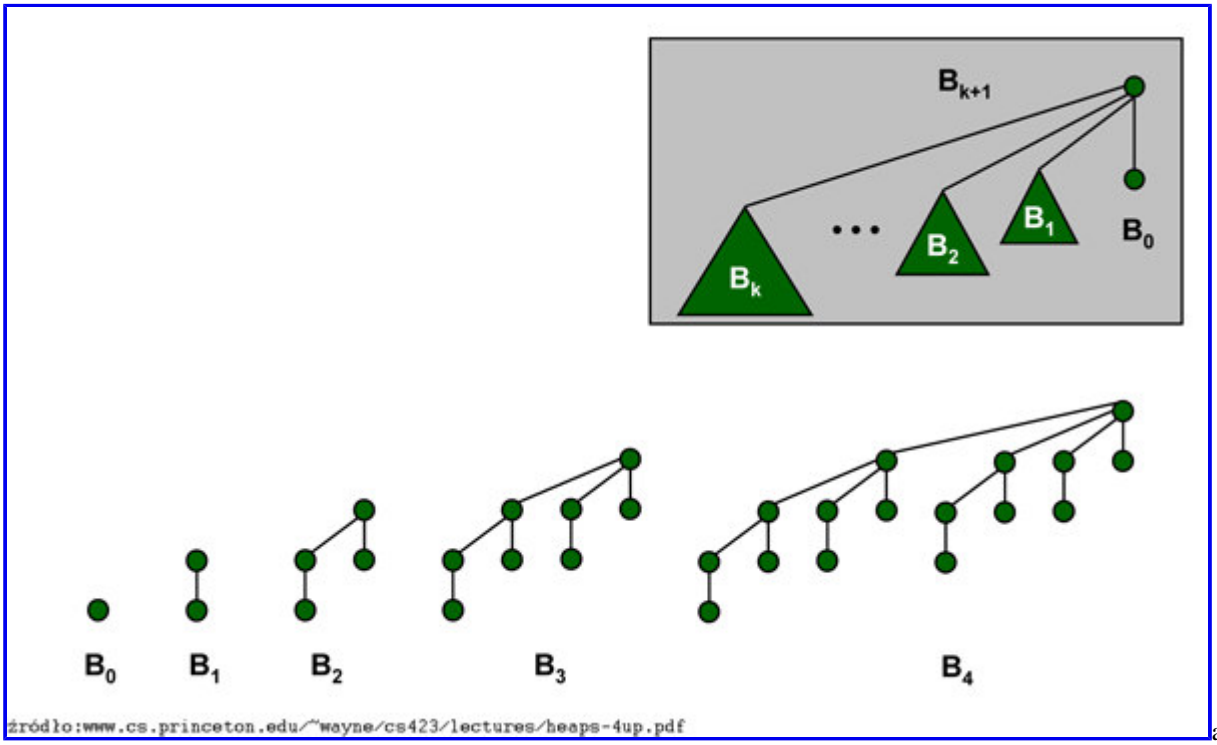
Złożoność obliczeniowa funkcji wstawiania nowego elementu do kopca czy usuwania z niego elementu maksymalnego (minimalnego w przypadku kolejek typu "min"; lub też dowolnego elementu) wynosi **O(lg n)**, gdzie n oznacza liczbę elementów w kolejce priorytetowej. Zatem czas potrzebny do wykonania tych obliczeń jest zadowalający.

W sytuacji, gdy do operowania na kolejkach priorytetowych ma być zaimplementowana metoda łączenia kolejek, kopiec binarny (czyli ten zwykły, który poznaliśmy) nie jest dobrą strukturą danych do zastosowania. Czas wykonania wspomnianej metody dla zwykłego kopca jest w pesymistycznym przypadku rzędu **O(n)**, a więc fakt ten zachęca do poszukiwania bardziej odpowiedniej struktury, której moglibyśmy użyć przy łączeniu kolejek.

Jedną z takich struktur jest kopiec dwumianowy. Czas wykonania operacji łączenia 2 stert wynosi O(lg n), czyli znacznie krócej niż w przypadku użycia zwykłej sterty binarnej (oczywiście różnica jest szczególnie widoczna dla dużych wartości n). Metodę scalania dwóch kolejek priorytetowych najczęściej oznaczamy jako **Union(S1, S2)** - w wyniku jej wywołania zostaje utworzona nowa kolejka zawierająca wszystkie elementy z S1 i S2, a obie kolejki „źródłowe" zostaną usunięte z pamięci.

Kopiec dwumianowy jest zbiorem drzew dwumianowych. Należy więc wyjaśnić, jak tworzone są takie drzewa. Mianowicie zasada ich budowy jest związana z zastosowaniem rekurencji:

- drzewo $B_0$ składa się z pojedynczego elementu
- drzewo $B_1$ jest połączeniem 2 drzew $B_0$
- ...
- drzewo $B_k$ jest połączeniem 2 drzew $B_{k-1}$

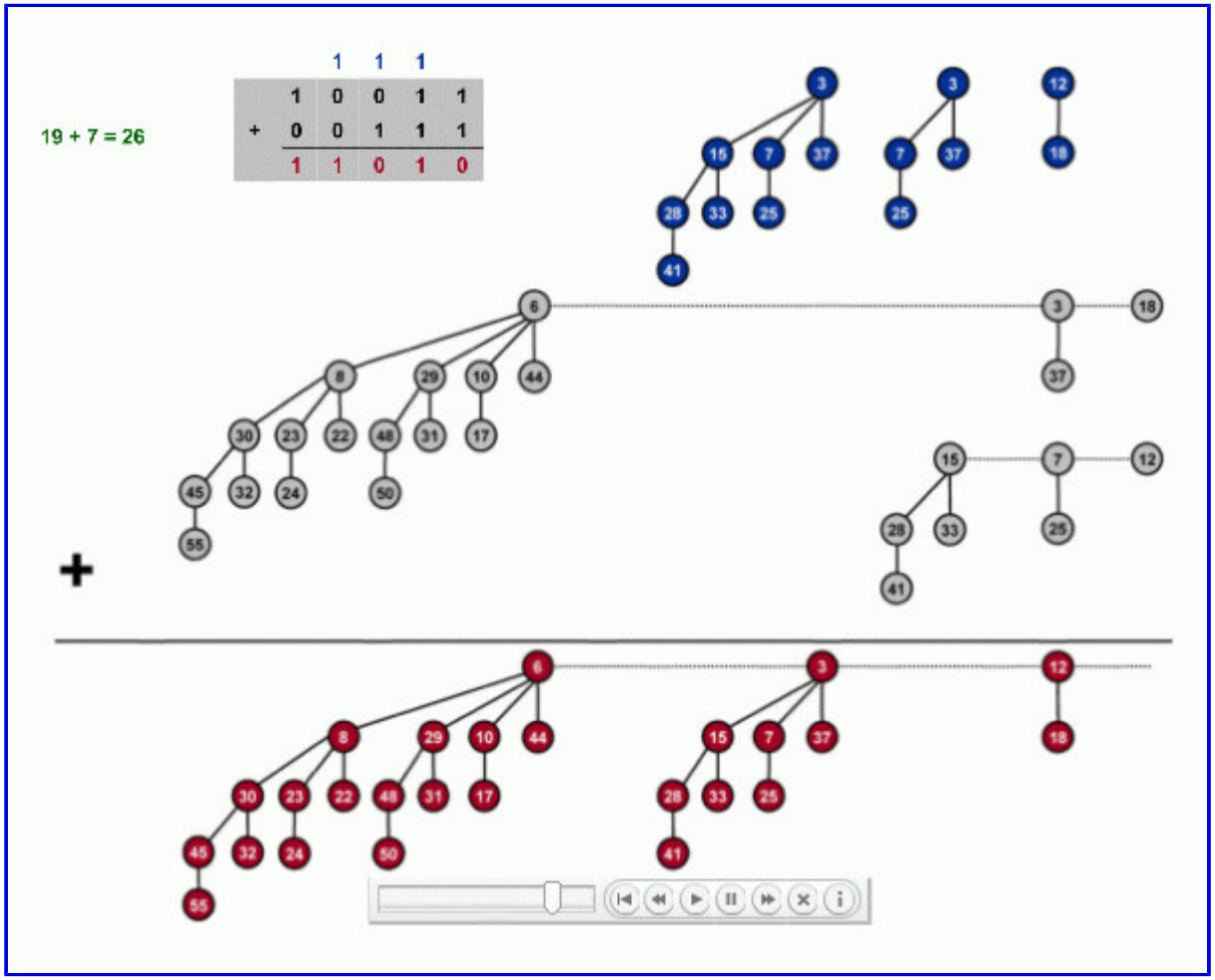


a>

*źródło:*
*wazniak.mimuw.edu.pl/index.php?title=Algorytmy_i_struktury_danych/Kolejki_priorytetowe*

Dwa drzewa stopnia k-tego są łączone w taki sposób, że korzeń jednego z nich staje się skrajnie lewym synem drugiego drzewa – w wyniku otrzymujemy drzewo o stopniu k+1. Każde z drzew posiada wszystkie właściwości zwykłej sterty. Natomiast kopiec dwumianowy jest kolekcją drzew dwumianowych, w której znajduje się co najwyżej jedno drzewo o danym stopniu.

W tym momencie wyjaśniliśmy podstawy budowania kopca dwumianowego. W celu prześledzenia sposobu, w jaki kopce są łączone, i uzupełnienia wiedzy w tym zakresie, zachęcamy do portalu ***wazniak.mimuw.edu.pl***, w którym omawiane są kolejki priorytetowe typu min (bliźniacze w stosunku do omawianych w naszym podręczniku). Żeby przejść do interesującego nas wykładu, należy kliknąć na poniższy rysunek (przedstawiający działanie metody *Union(S1,S2)* ).




Pozostałe typowe operacje obsługujące kolejki priorytetowe, takie jak *Insert, DelMin* czy *Del* (usunięcie dowolnego elementu) wykonują się przy zastosowaniu kopca dwumianowego w czasie O(lg n), czyli mają taką samą złożoność obliczeniową jak ich odpowiedniki zastosowane dla zwykłej sterty binarnej. Jedynie funkcja znajdowania minimalnego elementu posiada w tym przypadku większą złożoność obliczeniową - za sprawą konieczności przejścia przez wszystkie korzenie drzew dwumianowych należących do kopca, a następnie porównania ich wartości i określenia największego z nich. Taki sposób realizacji funkcji *FindMin* powoduje zwiększenie złożoności z O(1) (natychmiastowe wskazanie elementu minimalnego w kopcu binarnym) do O(lg n) - co jednak nie zniechęca do zastosowania kopca dwumianowego w sytuacji, gdy spodziewamy się częstej konieczności łączenia stert.

## Kopce Fibonacciego

Jednym z ulepszeń kopców dwumianowych jest struktura, która przyjęła nazwę kopców Fibonacciego. Pierwsza publikacja w czasopiśmie naukowym, opisująca wspomniane kopce, nastąpiła stosunkowo niedawno, w roku 1987. Jej autorami i zarazem twórcami pojęcia kopców Fibonacciego są Fredman oraz Tarjan.

Główną zaletą wykorzystywania takich kopców jako kolejek priorytetowych jest redukcja czasu wykonania operacji niezwiązanych z usuwaniem elementu z kolejki (czyli *Insert, FindMin, Union* czy *DecreaseKey*) w sensie zamortyzowanym do złożoności stałej O(1).

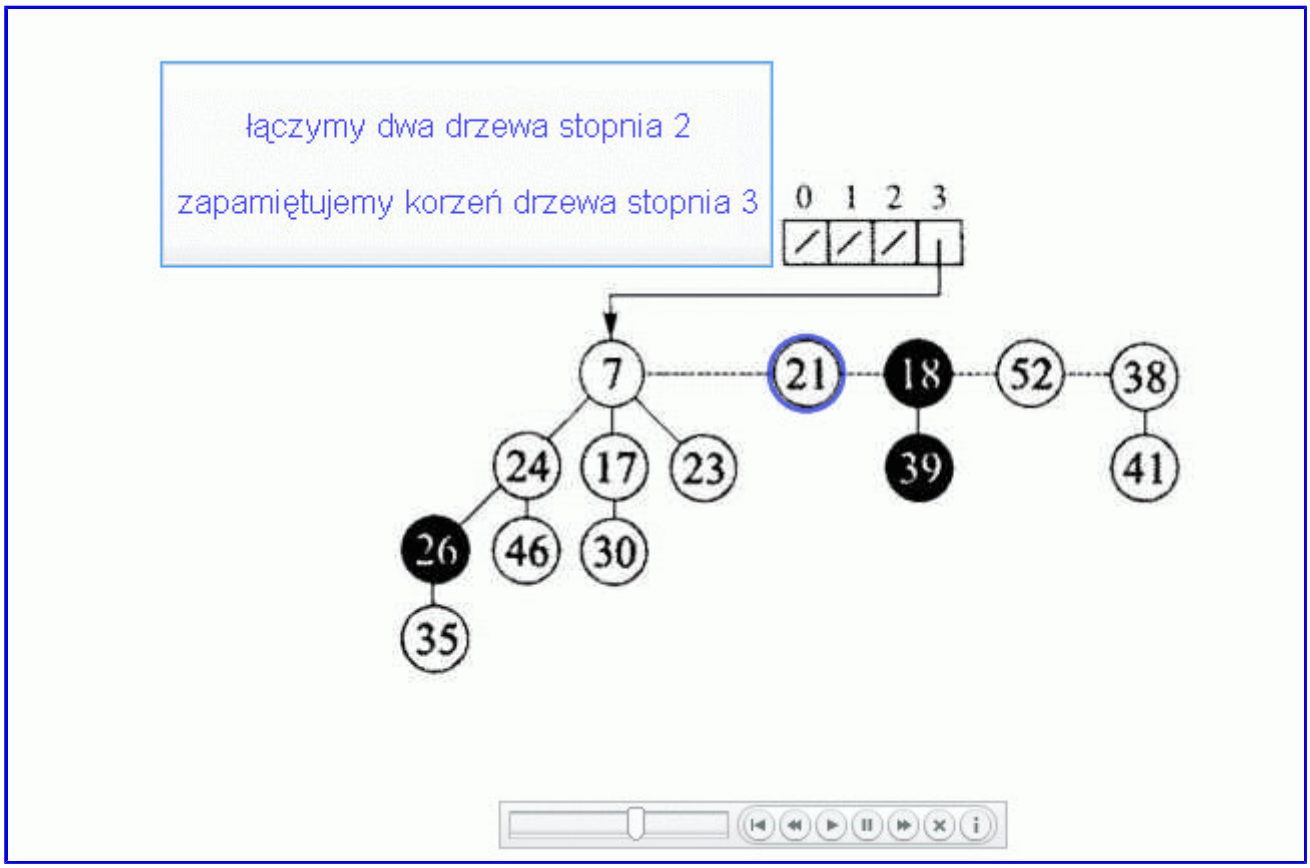
Pojęcie **kosztu zamortyzowanego** oznacza w tym przypadku średni czas wykonania operacji w najgorszym przypadku. Wartość takiego czasu może być stosunkowo mała, gdy dzielimy całkowity koszt (czas) wykonania zestawu operacji przez ich liczbę – w takiej sytuacji koszt pojedynczych operacji (niezależnie od ich rodzaju) jest jednakowy (taki przypadek ma miejsce, gdy stosujemy do amortyzacji metodę kosztu sumarycznego, inne metody amortyzacji pozwalają na ustalenie różnych kosztów dla odmiennych typów operacji). Zainteresowanych tematyką analizy kosztu zamortyzowanego odsyłamy do książki *T.Cormen, C. Leiserson, R. Rivest, C. Stein, Wprowadzenie do algorytmów, WNT 2004, rozdz. 17.*

Natychmiastowy czas wykonania wspomnianych wyżej operacji przeprowadzanych na kopcach Fibonacciego wynika z jednej podstawowej przyczyny – zostały one uproszczone do minimum. Wszystkie czynności związane z utrzymaniem poprawności struktury kopca Fibonacciego zostały odłożone „na później", gdy już będzie niezbędne ich przeprowadzenie – aby wszystkie funkcje kolejki priorytetowej działały poprawnie.

Kopce Fibonacciego, podobnie jak kopce dwumianowe, są zbiorem drzew, których korzenie są połączone w listę. Każde z drzew składowych spełnia własność zwykłego kopca (choć nie są to drzewa ani pełne, ani dwumianowe). Operacje łączenia kolejek (dodawanie elementu do kolejki jest również przypadkiem łączenia) wykonuje się poprzez zwykłe „sklejenie" dwóch kopców, nie przeprowadza się przy tym żadnego przemieszczania węzłów w drzewach. Dopiero w momencie, gdy chcemy usunąć jeden z

węzłów kopca, następuje uporządkowanie struktury całej kolejki priorytetowej.

W serwisie **wazniak.mimuw.edu.pl**, w rozdziale omawiającym kolejki priorytetowe, zostały w przystępny sposób omówione i zobrazowane algorytmy wykonujące się na kopcach Fibonacciego:




Wydawać by się mogło, że kopce Fibonacciego mogą być niezmiernie użyteczne w projektach informatycznych, w których usuwanie elementów z kolejki priorytetowej występuje rzadko w porównaniu z innymi metodami operującymi na kolejkach. Tymczasem okazuje się, że ze względu na bardzo złożoną implementację kopców Fibonacciego, ich wykorzystanie w praktyce nie jest najpopularniejsze - nadal częściej stosowane są kopce binarne.