O'REILLY

Nauka algorytmów

Poradnik pisania lepszego kodu

Helion

George Heineman

Tytuł oryginału: Learning Algorithms: A Programmer's Guide to Writing Better Code

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-8799-7

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/naualg.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/naualg
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 4.

Wędrówka po kopcu

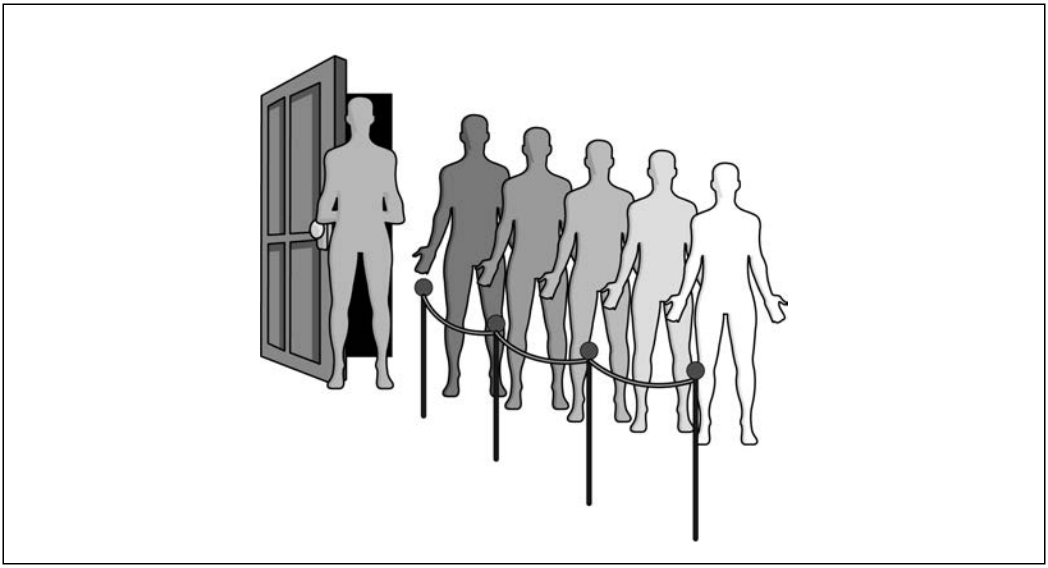
W tym rozdziale poznasz następujące zagadnienia:

- Typy danych *kolejka* i *kolejka priorytetowa*.
- Wymyślona w 1964 r. struktura danych o nazwie *kopiec binarny*, którą można zapisać w tablicy.
- W *kopcu binarnym typu max* element o *wyższej wartości priorytetu* jest uznawany za bardziej priorytetowy niż element o niższej wartości priorytetu. W *kopcu binarnym typu min* wyższy priorytet mają elementy o niższych wartościach priorytetu.
- Kolejkowanie elementów (wartość, priorytet) w kopcu binarnym w czasie O(`log` N), gdzie N to liczba elementów w kopcu.
- Znajdowanie w kopcu binarnym wartości o najwyższym priorytecie w czasie O(1).
- Usuwanie wartości o najwyższym priorytecie z kopca binarnego w czasie O(`log` N).

Co się stanie, jeśli zamiast zapisywać samą kolekcję wartości, zachowasz kolekcję elementów, z których każdy obejmuje *wartość* i powiązany z nią *priorytet* reprezentowany przez liczbę? Gdy dane są dwa elementy, ten o wyższym priorytecie jest ważniejszy niż drugi. Tym razem wyzwanie polega na tym, by umożliwić wstawianie nowych elementów (wartość, priorytet) do kolekcji oraz *usuwanie i zwracanie wartości elementu o najwyższym priorytecie*.

Tak działają *kolejki priorytetowe* — typ danych, który umożliwia wydajne wykonywanie operacji `enqueue(wartość, priorytet)` i `dequeue()` (usuwa wartość o najwyższym priorytecie). Kolejka priorytetowa różni się od opisanej w poprzednim rozdziale tablicy symboli, ponieważ *nie trzeba z góry znać priorytetu*, aby zażądać usunięcia wartości o najwyższym priorytecie.

Gdy popularny klub nocny staje się zbyt zatłoczony, na zewnątrz tworzy się kolejka, co ilustruje rysunek 4.1. Kiedy kolejne osoby chcą dostać się do klubu, muszą stanąć na końcu kolejki. Jako pierwsza osoba wejdzie do klubu ta, która czeka najdłużej. Tak działa ważny abstrakcyjny typ danych *kolejka*. Udostępnia ona operację `enqueue(wartość)`, która dodaje `wartość` jako najnowszy element na końcu kolejki, oraz operację `dequeue()`, usuwającą z kolejki najstarszą wartość. Jest to model FIFO (ang. *first in, first out*, czyli pierwszy na wejściu, pierwszy na wyjściu).

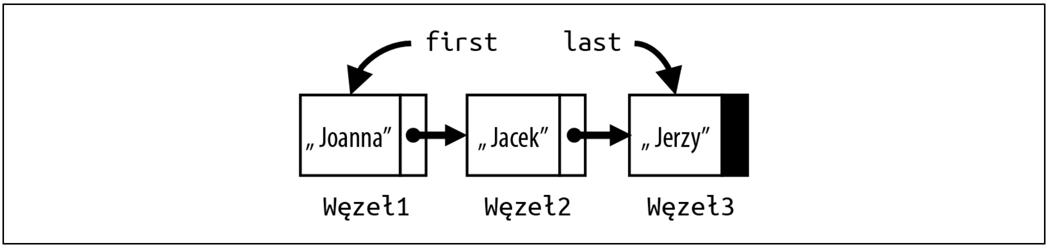


Rysunek 4.1. Oczekiwanie w kolejce w klubie nocnym

W poprzednim rozdziale opisałem listę powiązaną. Teraz użyjesz jej ponownie razem z klasą `Node`, która posłuży do przechowywania wartości (pole `value`) z kolejki:

```
class Node:
  def __init__(self, val):
    self.value = val
    self.next = None
```

Implementacja klasy `Queue` z listingu 4.1 używa tej struktury i udostępnia operację `enqueue()`, która dodaje wartość na koniec listy powiązanej. Rysunek 4.2 przedstawia wynik dodania wartości „Joanna”, „Jacek” i „Jerzy” (w takim porządku) do kolejki przed klubem nocnym.



Rysunek 4.2. Model kolejki przed klubem nocnym z trzema węzłami

„Joanna” jest pierwszym gościem pobieranym z kolejki. W kolejce pozostają wtedy dwie osoby, a pierwszą z nich jest „Jacek”.

W klasie `Queue` operacje `enqueue()` i `dequeue()` są wykonywane w stałym czasie, niezależnie od łącznej liczby wartości w kolejce.

Listing 4.1. Implementacja typu danych Queue bazująca na liście powiązanej

```
class Queue:
  def __init__(self):
    self.first = None         ❶
    self.last = None

  def is_empty(self):
    return self.first is None    ❷

  def enqueue(self, val):
    if self.first is None:       ❸
      self.first = self.last = Node(val)
    else:
      self.last.next = Node(val) ❹
      self.last = self.last.next

  def dequeue(self):
    if self.is_empty():
      raise RuntimeError('Kolejka jest pusta.')

    val = self.first.value       ❺
    self.first = self.first.next ❻
    return val
```

❶ Początkowo pola `first` i `last` mają wartość `None`.

❷ Jeśli pole `first` ma wartość `None`, obiekt `Queue` jest pusty.

❸ Jeśli obiekt `Queue` jest pusty, do `first` i `last` należy przypisać nowo utworzony węzeł (obiekt typu `Node`).

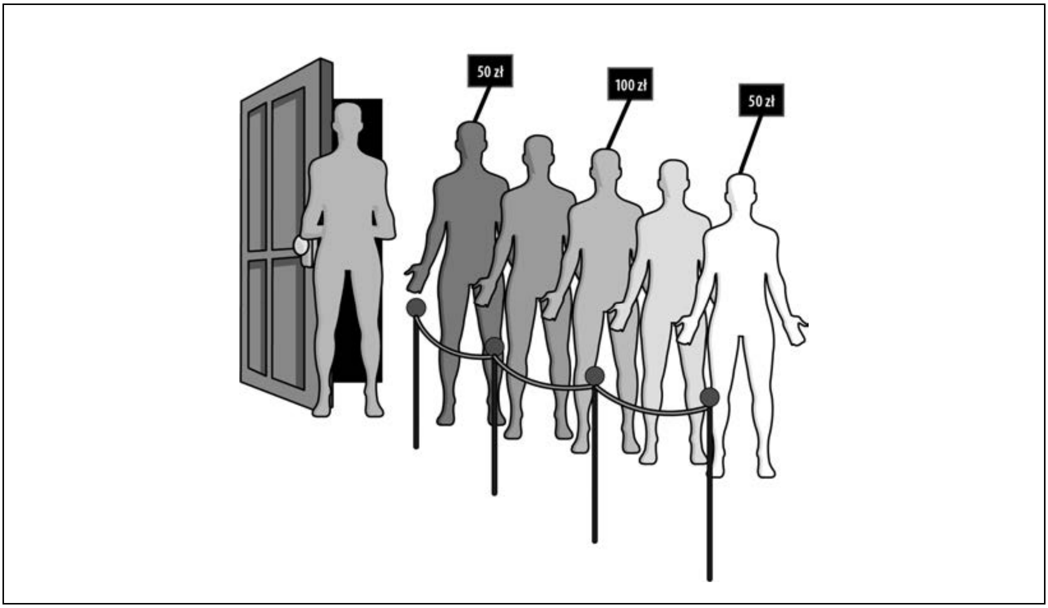
❹ Jeżeli obiekt `Queue` nie jest pusty, należy dodać element po `last` i zmodyfikować element `last`, aby wskazywał nowo utworzony węzeł (obiekt typu `Node`).

❺ Pole `first` wskazuje węzeł zawierający wartość, jaką należy zwrócić.

❻ Do `first` przypisywany jest drugi węzeł z listy (jeśli taki istnieje).

A oto inny scenariusz — klub nocny decyduje się umożliwić gościom zakup specjalnej wejściówki określającej kwotę do wydania. Na przykład jeden gość może kupić wejściówkę wartą 50 zł, a inny — wejściówkę za 100 zł. Gdy klub staje się zbyt zatłoczony, goście czekają w kolejce na wejście. Jednak jako pierwsza osoba do klubu zostanie wpuszczona ta, która *posiada najdroższą wejściówkę*. Jeśli dwie osoby mają równie drogie wejściówki, do klubu wejdzie jedna z nich. Osoby bez wejściówek są traktowane tak, jakby zapłaciły 0 zł.

Na rysunku 4.3 gość pośrodku, z wejściówką za 100 zł, wejdzie do klubu jako pierwszy. Potem zostaną wpuszczone dwie osoby z wejściówkami za 50 zł (w jakiejś kolejności). Wszystkie osoby bez wejściówek są traktowane tak samo, dlatego w dalszej kolejności każda z nich może zostać wpuszczona do klubu.



Rysunek 4.3. Goście mogą wejść szybciej dzięki kupionej wejściówce

> Kolejka priorytetowa nie określa, *co zrobić, gdy co najmniej dwie wartości mają ten sam najwyższy priorytet*. W niektórych implementacjach kolejka priorytetowa nie zwraca wartości w kolejności ich dodawania do kolejki. Opisana w tym rozdziale kolejka priorytetowa bazująca na kopcu nie zwraca wartości o tym samym priorytecie zgodnie z kolejnością ich zapisywania w kolejce. Wbudowany moduł `heapq` implementuje kolejkę priorytetową za pomocą kopca, co opisuję w rozdziale 8.

Ten zmodyfikowany model odpowiada abstrakcyjnemu typowi danych o nazwie *kolejka priorytetowa*. W tym typie danych nie da się wydajnie zaimplementować funkcji `enqueue()` i `dequeue()`, aby działały w stałym czasie. Jeśli używasz listy powiązanej, funkcja `enqueue()` nadal działa w czasie O(1), ale `dequeue()` może wymagać sprawdzenia wszystkich wartości w kolejce priorytetowej, aby znaleźć element o najwyższym priorytecie. Tak więc dla *przypadku pesymistycznego* złożoność tej funkcji wynosi O(N). Z kolei jeżeli zapiszesz wszystkie elementy posortowane według priorytetów, funkcja `dequeue()` będzie miała złożoność O(1), ale funkcja `enqueue()` dla *przypadku pesymistycznego* będzie miała złożoność O(N), co wynika z konieczności znalezienia miejsca wstawiania nowej wartości.

Na podstawie dotychczasowych rozważań można zaproponować pięć struktur, w których do zapisywania elementów (wartość, priorytet) używane są obiekty typu `Entry`:

Tablica

Jest to *tablica nieposortowanych elementów*, która nie ma określonej struktury i wymaga liczenia na szczęście. Funkcja `enqueue()` działa tu w czasie stałym, ale funkcja `dequeue()` musi przeszukać całą tablicę, aby znaleźć wartość o najwyższym priorytecie do usunięcia i zwrócenia. Ponieważ tablica ma stałą wielkość, taka kolejka priorytetowa może się zapełnić.

Wbudowane operacje

Używana jest tu *lista nieposortowana*, dla której działają wbudowane operacje Pythona. Wydajność jest tu podobna do wydajności dla *tablicy*.

Posortowana tablica

Tablica zawierająca elementy posortowane rosnąco według priorytetów. W funkcji `enqueue()` *używana jest odmiana wyszukiwania binarnego w tablicy (z listingu 2.4) do znajdowania miejsca, w którym należy umieścić element. Następnie trzeba ręcznie przesunąć elementy tablicy, aby zrobić miejsce na nową wartość. Funkcja* `dequeue()` *działa w stałym czasie, ponieważ elementy są posortowane, a wartość o najwyższym priorytecie znajduje się na końcu tablicy.* Ponieważ tablica ma stałą wielkość, taka kolejka priorytetowa może się zapełnić.

Lista powiązana

Jest to lista powiązana elementów, na której pierwszy element ma najwyższy priorytet z wszystkich wartości, a każdy kolejny ma priorytet nie większy od poprzedniego. W tej implementacji nowe wartości są dodawane do kolejki w odpowiednim miejscu listy powiązanej, dzięki czemu funkcja `dequeue()` *działa w stałym czasie.*

Posortowana lista

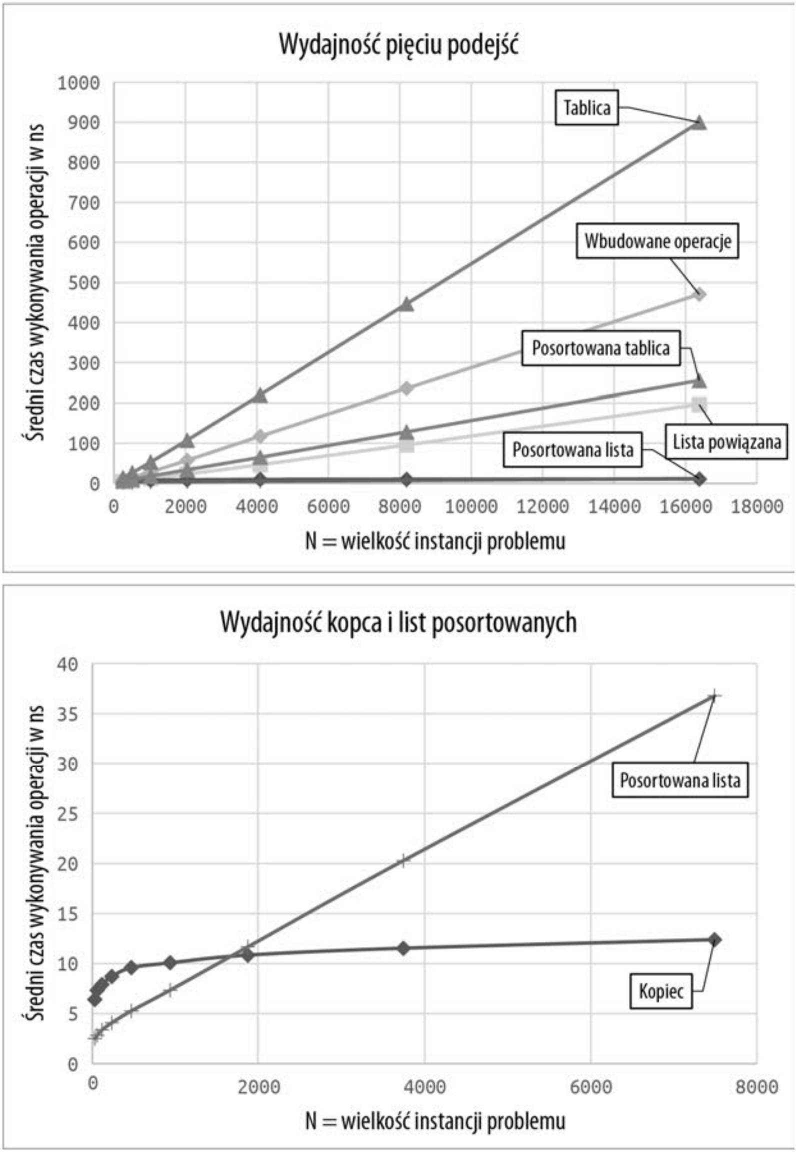
Jest to lista Pythona zawierająca elementy posortowane rosnąco według priorytetów. W funkcji `enqueue()` *używana jest odmiana wyszukiwania binarnego w tablicy do dynamicznego umieszczania elementów w odpowiednim miejscu. Funkcja* `dequeue()` *działa w stałym czasie, ponieważ element o najwyższym priorytecie zawsze znajduje się na końcu listy.*

Aby porównać te implementacje, zaprojektowałem eksperyment, który wykonuje 3N/2 operacji `enqueue()` i 3N/2 operacji `dequeue()`. W każdej implementacji mierzony jest łączny czas wykonywania, który dzielony jest przez 3N, aby obliczyć średni koszt operacji. W tabeli 4.1 pokazane jest, że tablica o stałej wielkości daje najgorsze wyniki, a wbudowane listy Pythona pozwalają o połowę skrócić czas. Tablica posortowanych elementów pozwala przyspieszyć pracę jeszcze o połowę, a listy powiązane powodują dalszy wzrost wydajności o 20%. Zdecydowanym zwycięzcą jest jednak posortowana *lista*.

Tabela 4.1. Średnia wydajność operacji (czas w ns) dla instancji problemu o wielkości N

N	Kopiec	Posortowana lista	Lista powiązana	Posortowana tablica	Wbudowane listy	Tablica
256	6,4	2,5	3,9	6,0	8,1	13,8
512	7,3	2,8	6,4	9,5	14,9	26,4
1024	7,9	3,4	12,0	17,8	28,5	52,9
2048	8,7	4,1	23,2	33,7	57,4	107,7
4096	9,6	5,3	46,6	65,1	117,5	220,3
8192	10,1	7,4	95,7	128,4	235,8	446,6
16 384	10,9	11,7	196,4	255,4	470,4	899,9
32 768	11,5	20,3	—	—	—	—
65 536	12,4	36,8	—	—	—	—

W wymienionych technikach średni koszt wykonywania operacji `enqueue()` i `dequeue()` rośnie wprost proporcjonalnie do N. W kolumnie *Kopiec* w tabeli 4.1 pokazana jest wydajność dla struktury danych Heap. Średni koszt rośnie tu proporcjonalnie do `log`(N), co ilustruje rysunek 4.4. Dlatego implementacja używająca tej struktury jest znacznie wydajniejsza od implementacji bazującej na posortowanych listach Pythona. Na złożoność logarytmiczną wskazuje stały wzrost czasu działania przy podwajaniu się wielkości problemu. W tabeli 4.1 widać, że każde podwojenie wielkości problemu skutkuje wzrostem czasu o ok. 0,8 nanosekundy.



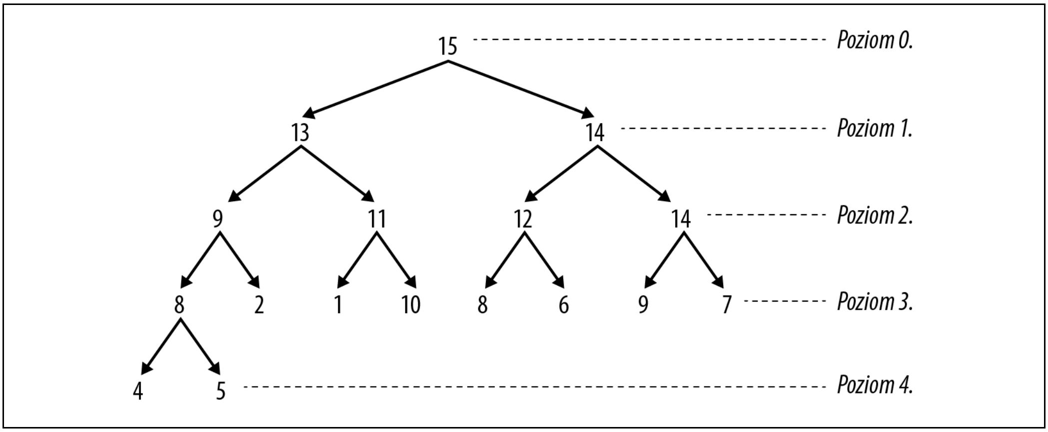
Rysunek 4.4. Złożoność O(log N) kopca (struktura Heap) jest lepsza niż złożoność O(N) pozostałych technik

Struktura danych o nazwie *kopiec* została wymyślona w 1964 roku. Operacje na kolejce priorytetowej wykonywane z użyciem kopca mają złożoność O(`log` N). W tym rozdziale nie są istotne wartości dodawane do kolejki. Mogą to być łańcuchy znaków, liczby, a nawet grafiki — nie ma to znaczenia. Istotne są tylko wartości priorytetu każdego elementu. W każdym z pozostałych rysunków w tym rozdziale pokazane są tylko priorytety elementów dodanych do kolejki. Gdy dane są dwa elementy w kopcu typu max, ten o priorytecie o większej wartości ma wyższy priorytet.

Kopiec ma znaną od początku wielkość maksymalną M i może pomieścić N < M elementów. Teraz omówię strukturę kopca, pokażę, że może zwiększać się i zmniejszać (w granicach wielkości maksymalnej), a także wyjaśnię, jak N elementów kopca jest zapisywanych w zwykłej tablicy.

Kopce binarne typu max

Ten pomysł może wydawać się dziwny, ale co się stanie, jeśli tylko „częściowo posortujesz” elementy? Rysunek 4.5 ilustruje kopiec typu max zawierający 17 elementów. Dla każdego elementu *pokazany jest tylko priorytet*. Widać tu, że na poziomie 0 znajduje się jeden element; ma on najwyższy priorytet spośród wszystkich elementów kopca typu max. Strzałka w $x \rightarrow y$ oznacza, że priorytet elementu x jest większy lub równy względem priorytetu elementu y.



Rysunek 4.5. Przykładowy kopiec binarny typu max

Te elementy nie są w pełni posortowane, jak byłyby posortowane na liście, dlatego trzeba chwilę poszukać, aby znaleźć element o najniższym priorytecie (wskazówka — znajduje się on na poziomie 3.). Jednak uzyskana struktura ma kilka przydatnych cech. Na poziomie 1. znajdują się dwa elementy. Jeden z nich musi mieć drugi najwyższy priorytet (lub równy najwyższemu, prawda?). Każdy poziom `k` *oprócz ostatniego* jest *pełny* i zawiera 2^k elementów. Tylko najniższy poziom jest niepełny (tu ma 2 elementy z możliwych 16) i jest zapełniany od lewej do prawej. Na rysunku widać też, że ten sam priorytet może występować w kopcu wielokrotnie. W pokazanym kopcu powtarzają się priorytety 8 i 14.

Z każdego elementu wychodzą nie więcej niż dwie strzałki, dlatego jest to *kopiec binarny typu max*. Przyjrzyj się elementowi z poziomu 0. Element ten ma priorytet 15. Pierwszy element z poziomu 1. (o priorytecie 13) to *lewe dziecko*; drugi element z poziomu 1. (o priorytecie 14) to *prawe dziecko*. Element o priorytecie 15 to *rodzic* dwójki dzieci z poziomu 1.

Oto zestawienie cech kopca binarnego typu max:

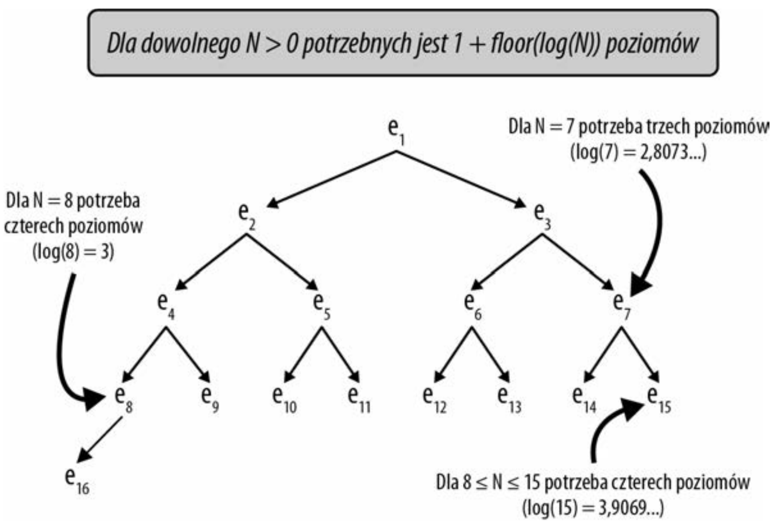
Uporządkowanie elementów kopca

Priorytet elementu jest większy lub równy względem priorytetów lewego i prawego dziecka (jeśli te istnieją). Priorytet każdego elementu (oprócz elementu z najwyższego poziomu) jest mniejszy lub równy względem priorytetu rodzica.

Struktura kopca

Każdy poziom `k` musi być zapełniony 2^k elementami (od lewej do prawej), zanim dodany zostanie jakikolwiek element na poziomie `k` + 1.

Gdy kopiec binarny zawiera tylko jeden element, występuje tylko jeden poziom, 0., ponieważ $2^0 = 1$. Ile poziomów jest potrzebnych w kopcu binarnym przechowującym N > 0 elementów? Należy zdefiniować matematyczny wzór `L`(N), który zwraca liczbę poziomów potrzebnych dla N elementów. Rysunek 4.6 przedstawia wizualizację pomagającą ustalić `L`(N). Widocznych jest tu 16 elementów, każdy oznaczony indeksem. Indeksowanie rozpoczyna się od góry (e_1), a indeksy rosną od lewej do prawej do momentu, gdy na danym poziomie nie ma już więcej elementów. Potem indeksowanie jest kontynuowane od pierwszego od lewej elementu na następnym poziomie.

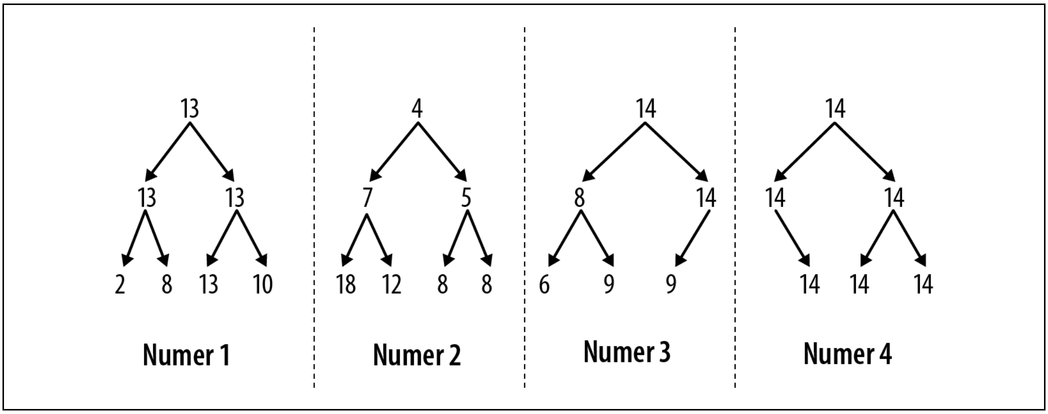


Rysunek 4.6. Określanie liczby poziomów potrzebnych w kopcu binarnym z N elementami

Gdy kopiec zawiera tylko siedem elementów, potrzebne są trzy poziomy z elementami od e_1 do e_7. Dla ośmiu elementów potrzebne są cztery poziomy. Jeśli przyjrzysz się lewym strzałkom, zobaczysz, że indeksy tworzą wzór — e_1, e_2, e_4, e_8 i e_{16}. Są to potęgi dwójki. Okazuje się, że `L`(N) = 1 + `floor`(`log`(N)).

Każdy nowy poziom w kopcu binarnym zawiera więcej elementów niż łączna liczba elementów na *wszystkich wcześniejszym poziomach*. Gdy zwiększysz wysokość kopca binarnego o jeden poziom, kopiec binarny będzie mieścił ponad dwukrotnie więcej elementów (w sumie 2N + 1, gdzie N to liczba istniejących elementów)!

Z omówienia logarytmów powinieneś pamiętać, że podwojenie N powoduje zwiększenie wartości `log`(N) o 1. Matematycznie jest to zapisywane tak — `log`(2N) = 1 + `log`(N). Które z czterech struktur z rysunku 4.7 są poprawnymi kopcami binarnymi typu max?



Rysunek 4.7. Które z tych struktur są poprawnymi kopcami binarnymi typu max?

Najpierw sprawdź kształt każdej z tych struktur. Wersje numer 1 i 2 są akceptowalne, ponieważ każdy poziom jest pełny. Struktura numer 3 też jest poprawna, ponieważ niepełny jest tylko ostatni poziom, który zawiera trzy (z możliwych czterech) elementy zapisane od lewej do prawej. Wersja numer 4 narusza właściwość struktury kopca, ponieważ ostatni poziom zawiera trzy elementy, ale brakuje w nim pierwszego elementu od lewej.

Teraz rozważ właściwość uporządkowania kopca binarnego typu max. Ta właściwość wymaga, aby priorytet każdego rodzica był większy lub równy względem priorytetów jego dzieci. Wersja numer 1 jest poprawna, o czym można się przekonać, sprawdzając każdą możliwą strzałkę. Wersja numer 3 jest nieprawidłowa, ponieważ element o priorytecie osiem ma prawe dziecko o wyższym priorytecie, dziewięć. Struktura numer 2 jest niepoprawna, ponieważ element o priorytecie cztery z poziomu 0. ma niższy priorytet niż oba elementy potomne.

Struktura numer 2 jest poprawnym przykładem *kopca binarnego typu min*, w którym priorytet każdego rodzica jest niższy lub równy względem priorytetów jego dzieci. Kopce binarne typu min zastosujesz w rozdziale 7.

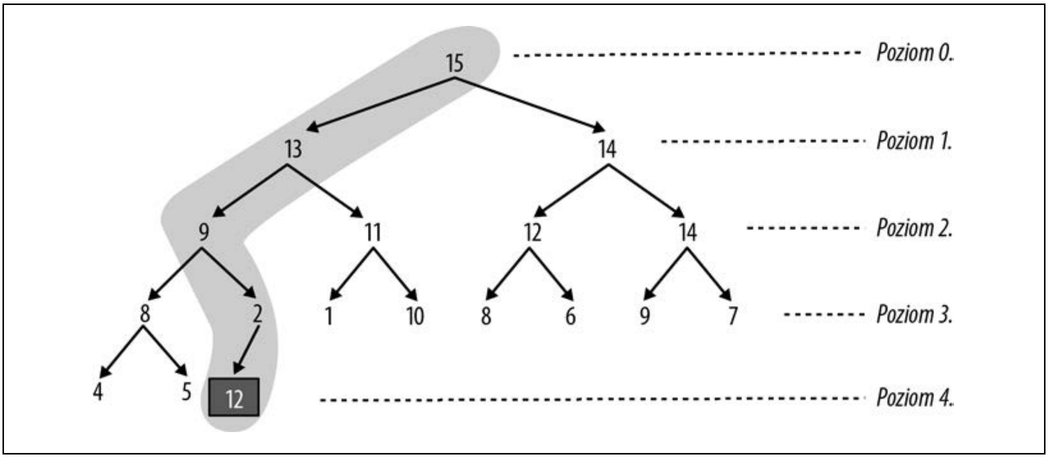
Należy zapewnić, że obie właściwości kopców zostaną zachowane po wykonaniu operacji `enqueue()` i `dequeue()` (ta druga usuwa element o najwyższym priorytecie z kopca typu max). Jest to ważne, ponieważ dalej wykażę, że obie te operacje działają w czasie O(`log` N), co oznacza znaczną poprawę względem technik przedstawionych wcześniej w tabeli 4.1, dla których funkcje `enqueue()` i `dequeue()` mają dla *przypadku pesymistycznego* złożoność O(N).

Wstawianie elementu (wartość, priorytet)

Gdzie należy wstawić nowy element po wywołaniu `enqueue(wartość, priorytet)` dla kopca binarnego typu max? Oto strategia, która zawsze się sprawdza:

- Umieść nowy element w pierwszej dostępnej pustej lokalizacji na ostatnim poziomie.
- Jeśli ten poziom jest pełny, powiększ kopiec o dodatkowy poziom i umieść nowy element na pozycji pierwszej od lewej na nowym poziomie.

Na rysunku 4.8 nowy element o priorytecie 12 jest wstawiany na trzeciej pozycji na poziomie 4. Właściwość struktury kopca zostaje zachowana, ponieważ elementy z niepełnego poziomu 4. rozpoczynają się od elementu pierwszego od lewej, a na poziomie tym nie występują luki. Może się jednak zdarzyć, że właściwość uporządkowania kopca została naruszona.

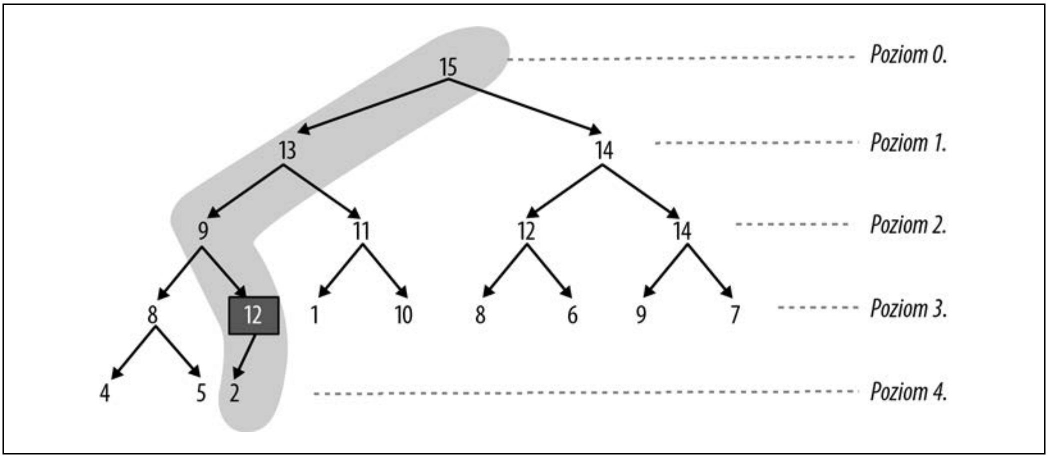


Rysunek 4.8. Pierwszy krok przy wstawianiu elementu polega na umieszczeniu go na następnej dostępnej pozycji

Dobra wiadomość jest taka, że wystarczy zmienić uporządkowanie elementów na *ścieżce* od nowo dodanej wartości do pierwszego elementu z poziomu 0. Na rysunku 4.10 pokazany jest efekt przywrócenia uporządkowania kopca. Widać tu, że elementy na wyróżnionej ciemniejszym kolorem ścieżce zostały odpowiednio przestawione, tak aby były uporządkowane malejąco od góry do dołu (kolejne elementy mogą też być równe).

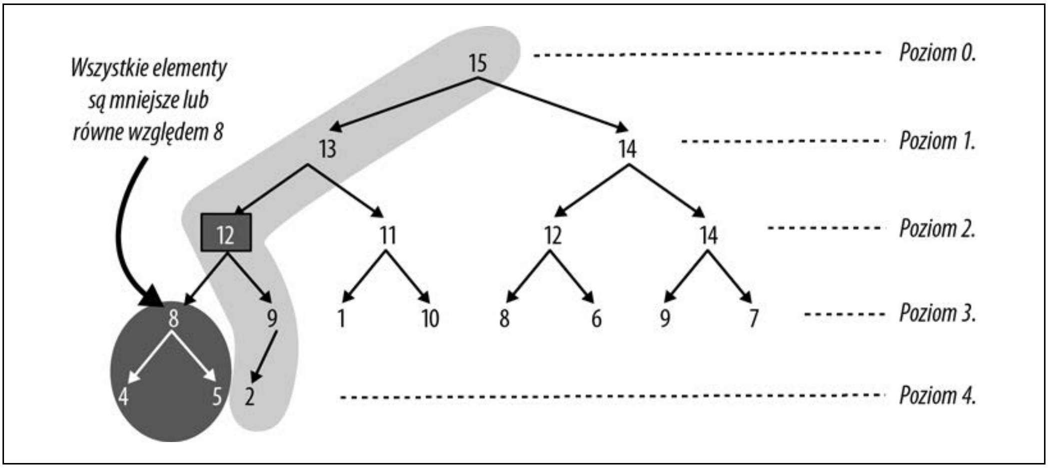
Ścieżka do określonego elementu w kopcu binarnym to sekwencja wyznaczona przez strzałki (lewe i prawe) od pojedynczego elementu z poziomu 0. do szukanego elementu.

W procesie modyfikowania kopca typu max w taki sposób, by zachowana została właściwość uporządkowania, nowo dodany element „wypływa” wzdłuż ścieżki do odpowiedniej lokalizacji w wyniku przestawiania par elementów. Na rysunku 4.8 widać, że nowo dodany element o priorytecie 12 narusza właściwość uporządkowania kopca, ponieważ ma priorytet większy niż priorytet rodzica (równy 2). *Przestaw te dwa elementy*, aby powstał kopiec typu max pokazany na rysunku 4.9, *i kontynuuj ten proces, idąc w górę*.



Rysunek 4.9. W drugim kroku element jest w razie potrzeby przenoszony o poziom w górę

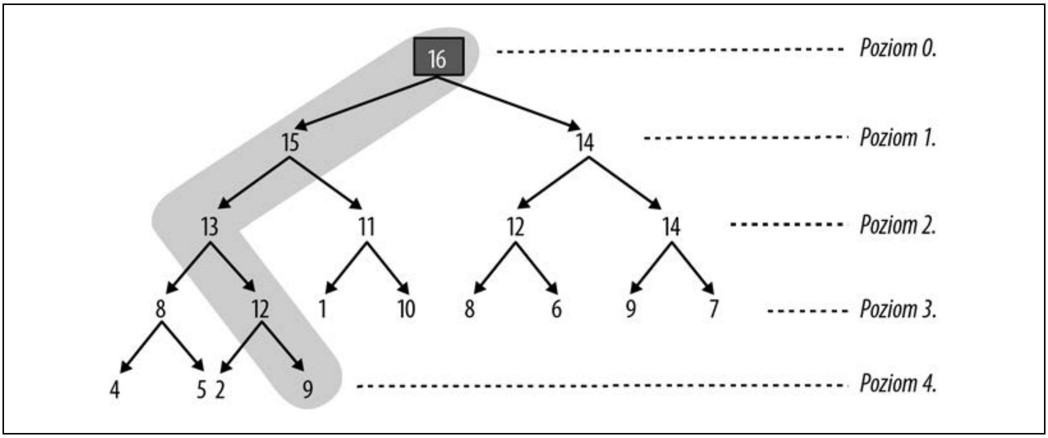
Od priorytetu 12 w dół wartości tworzą poprawny kopiec binarny typu max z dwoma elementami. Jednak element o priorytecie 12 nadal narusza właściwość uporządkowania, ponieważ jego rodzic ma niższy priorytet, 9. Przestaw więc nowy element z rodzicem, tak jak jest to pokazane na rysunku 4.10.



Rysunek 4.10. W trzecim kroku element jest w razie potrzeby przenoszony o poziom w górę

Na rysunku 4.10 od wyróżnionego elementu o priorytecie 12 w dół struktura jest poprawnym kopcem binarnym typu max. W momencie przestawiania elementów o priorytetach 9 i 12 nie musiałeś przejmować się elementami znajdującymi się poniżej elementu o priorytecie 8, *ponieważ wiadomo, że wszystkie one mają priorytety mniejsze lub równe względem 8*. Oznacza to, że mają one priorytet niższy niż 12. Ponieważ priorytet 12 jest mniejszy niż priorytet rodzica (13), właściwość uporządkowania kopca jest spełniona.

Spróbuj samodzielnie prześledzić operację `enqueue(wartość, 16)` w kopcu pokazanym na rysunku 4.10. Nowy element początkowo jest umieszczany na czwartej pozycji na poziomie 4. jako prawe dziecko elementu o priorytecie 9. Nowy element „wypływa" aż do poziomu 0., co daje w efekcie kopiec binarny typu max pokazany na rysunku 4.11.



Rysunek 4.11. Dodawanie elementu o priorytecie 16., który „wypływa" na samą górę

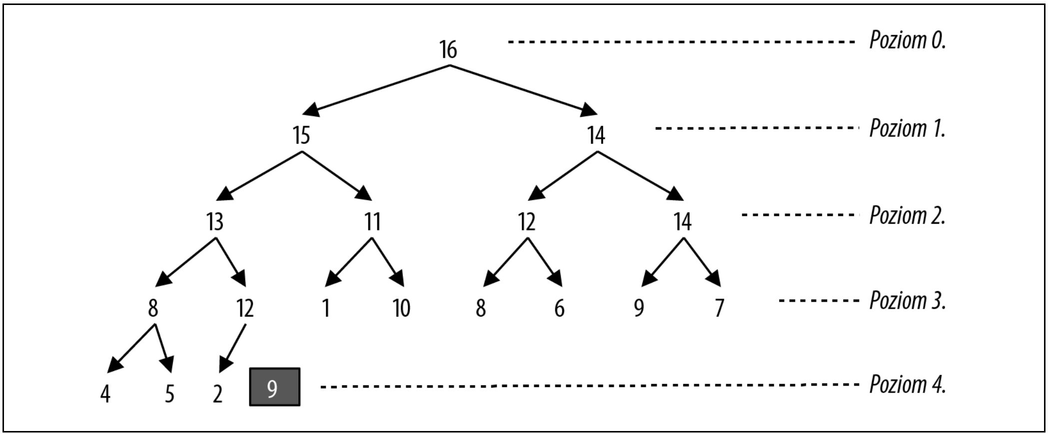
Przypadek pesymistyczny ma miejsce, gdy dodawany jest element o priorytecie wyższym niż innych elementów kopca binarnego typu max. Liczba elementów na ścieżce wynosi 1 + `floor(log`(N)), co oznacza, że *największa liczba przestawień* jest o jeden mniejsza i wynosi `floor(log`(N)). Teraz mogę jednoznacznie stwierdzić, że złożoność czasowa przebudowy kopca binarnego typu max w operacji `enqueue()` wynosi O(`log` N). Jednak ten świetny wynik dotyczy tylko połowy problemu, ponieważ dodatkowo trzeba zagwarantować wydajne usuwanie elementu o najwyższym priorytecie z kopca.

Usuwanie wartości o najwyższym priorytecie

Znajdowanie elementu o najwyższym priorytecie w kopcu binarnym typu max jest proste — zawsze jest to pojedynczy element z poziomu 0. Nie można jednak po prostu usunąć tego elementu, ponieważ naruszona zostanie struktura kopca — na poziomie 0. powstanie luka. Na szczęście istnieje strategia wykonywania operacji `dequeue()`, która pozwala usunąć górny element i wydajnie przebudować kopiec binarny typu max. Przedstawiam ją na następnych rysunkach:

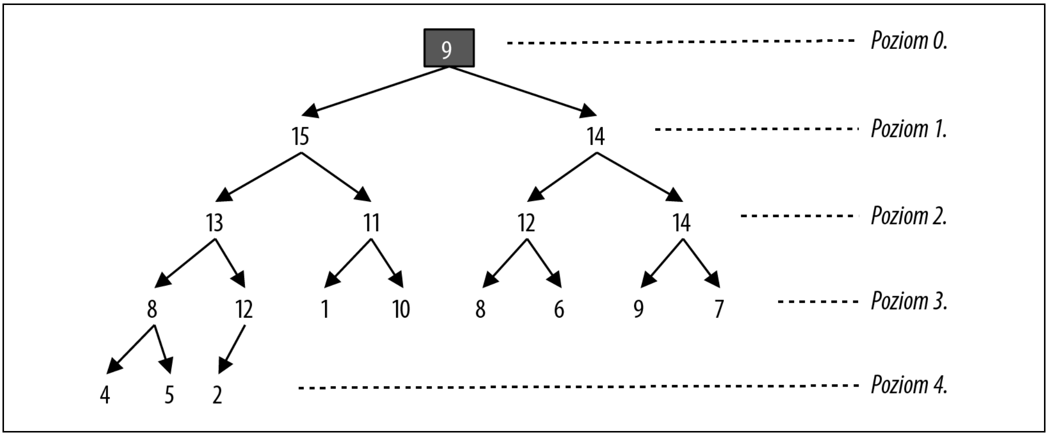
1. Usuń pierwszy od prawej element z dolnego poziomu i zapamiętaj go. Wynikowa struktura jest zgodna z właściwościami uporządkowania i struktury kopca.
2. Zapisz wartość elementu o najwyższym priorytecie (z poziomu 0.), aby można ją było zwrócić.
3. Zastąp element z poziomu 0. elementem usuniętym z dolnego poziomu kopca. Może to naruszyć właściwość uporządkowania kopca.

Aby wykonać zadanie, należy najpierw usunąć i zapamiętać element o priorytecie 9, co ilustruje rysunek 4.12. Wynikowa struktura nadal jest kopcem. Następnie zapisz wartość powiązaną z elementem o najwyższym priorytecie z poziomu 0., aby można ją było zwrócić (nie jest to pokazane na rysunku).



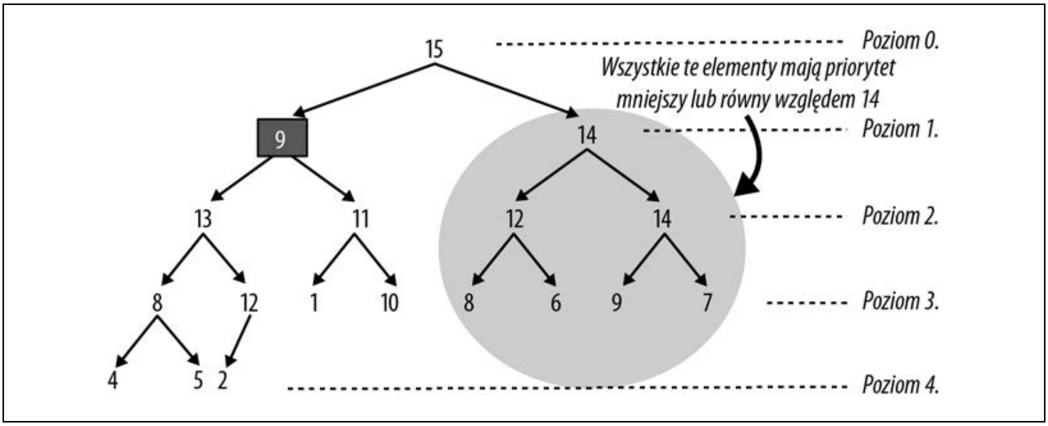
Rysunek 4.12. Pierwszy krok polega na usunięciu ostatniego elementu z dolnego poziomu

W ostatnim kroku zastąp pojedynczy element z poziomu 0. usuniętym elementem. Powstanie nieprawidłowy kopiec widoczny na rysunku 4.13. Widać tu, że priorytet jedynego elementu z poziomu 0. nie jest większy niż priorytety jego lewego dziecka (15) i prawego dziecka (14). Aby naprawić ten kopiec, musisz „zatopić" przeniesiony element, by znalazł się na pozycji na niższym poziomie kopca. Trzeba to zrobić tak, aby właściwość uporządkowania kopca ponownie była spełniona.



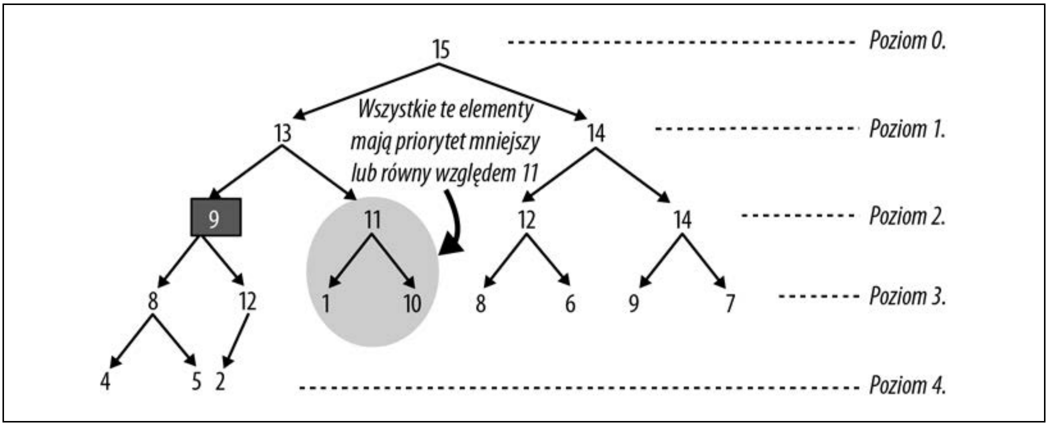
Rysunek 4.13. Nieprawidłowy kopiec powstały w wyniku wstawienia ostatniego elementu na poziomie 0.

Zacznij od błędnego elementu (czyli elementu o priorytecie 9 z poziomu 0.). Ustal, które z jego dzieci (lewe czy prawe) ma wyższy priorytet. Jeśli istnieje tylko lewe dziecko, uwzględnij tylko je. W tym przykładzie lewe dziecko ma wyższy priorytet (15) niż prawe (14). Rysunek 4.14 przedstawia efekt przestawienia elementu z poziomu 0. z dzieckiem o wyższym priorytecie.



Rysunek 4.14. Przestawienie górnego elementu z jego lewym dzieckiem (jest to dziecko o wyższym priorytecie)

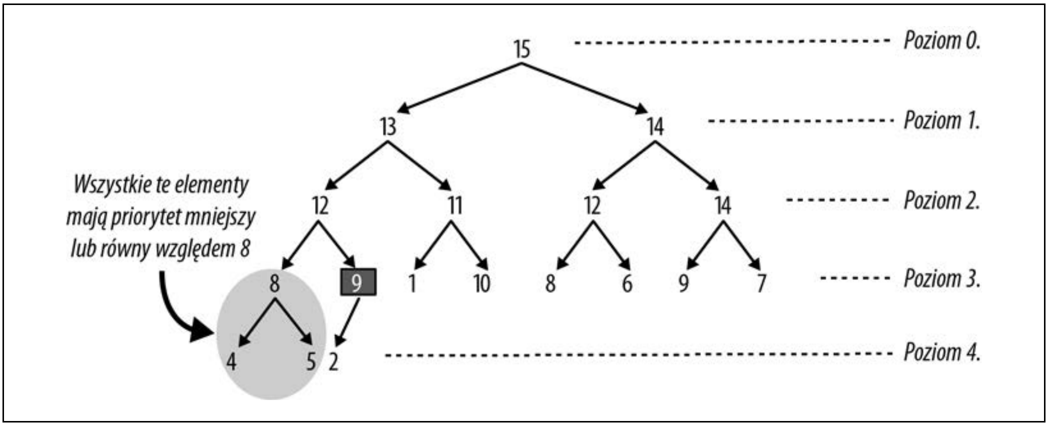
Widać tu, że cała podstruktura rozpoczynająca się od elementu o priorytecie 14 z poziomu 1. jest poprawnym kopcem binarnym typu max, dlatego nie wymaga zmian. Jednak świeżo przestawiony element o priorytecie 9. narusza właściwość uporządkowania kopca (ma mniejszy priorytet niż każde z jego dzieci). Dlatego należy kontynuować „zatapianie" go, tym razem po lewej stronie, ponieważ lewy element ma wyższy priorytet (13) z dwójki dzieci. Ilustruje to rysunek 4.15.



Rysunek 4.15. „Zatapianie" elementu o dodatkowy poziom

To już prawie koniec! Na rysunku 4.15 widać, że wśród dzieci elementu o priorytecie 9 wyższy priorytet (12) ma dziecko prawe. Dlatego przestaw odpowiednie elementy, co ostatecznie pozwoli spełnić właściwość uporządkowania kopca. Przedstawia to rysunek 4.16.

Inaczej niż przy dodawaniu nowego elementu do kolejki priorytetowej, tu nie ma prostej ścieżki przestawionych elementów, jednak mimo to można ustalić maksymalną liczbę powtórzeń operacji „zatapiania". Jest ona o jeden mniejsza niż liczba poziomów w kopcu binarnym typu max i wynosi `floor(log(N))`.



Rysunek 4.16. Wynikowy kopiec po „zatopieniu” elementu, by dotarł do prawidłowej lokalizacji

Możesz też zliczyć porównania priorytetów dwóch elementów. W każdy kroku „zatapiania” potrzebne są maksymalnie dwa porównania — jedno w celu znalezienia większego priorytetu wśród dzieci i jedno w celu stwierdzenia, czy priorytet rodzica jest wyższy niż większy z priorytetów dzieci. Oznacza to, że w sumie liczba porównań jest nie większa niż 2 · `floor(log(N))`.

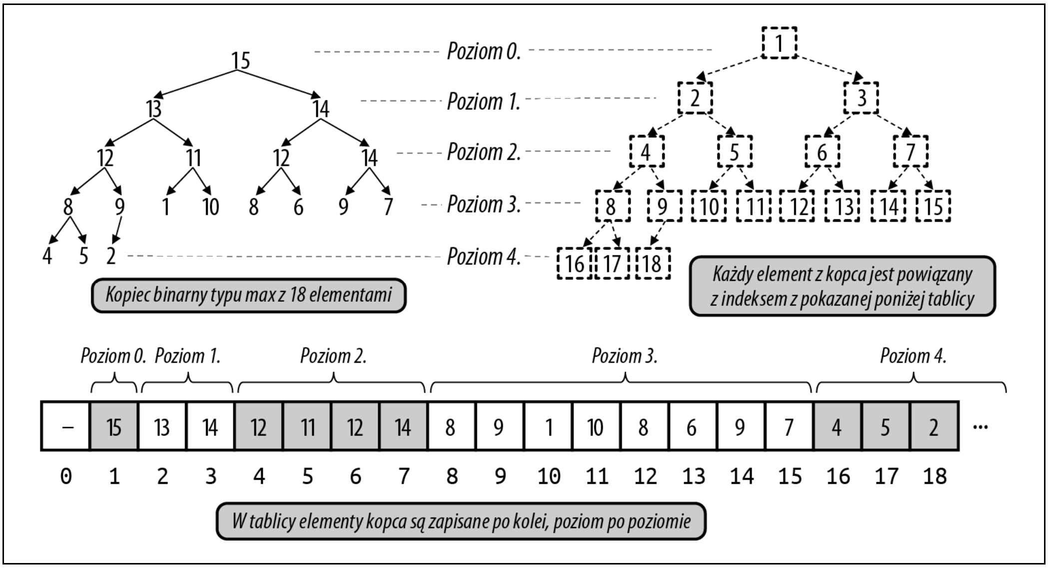
Bardzo ważne jest to, że w *przypadku pesymistycznym* kopiec binarny typu max umożliwia dodawanie elementu i usuwanie elementu o najwyższym priorytecie w czasie wprost proporcjonalnym do `log(N)`. Teraz pora przejść od teorii do praktyki i pokazać, jak zaimplementować kopiec binarny z użyciem zwykłej tablicy.

Czy zauważyłeś, że właściwość struktury kopca gwarantuje, iż można wczytać wszystkie elementy po kolei od lewej do prawej, począwszy od poziomu 0. przez każdy kolejny poziom? Można wykorzystać to spostrzeżenie i zapisać kopiec binarny w zwykłej tablicy.

Reprezentowanie kopca binarnego za pomocą tablicy

Rysunek 4.17 pokazuje, jak zapisać kopiec binarny typu max z N = 18 elementami w stałej tablicy o wielkości M > N. Ten pięciopoziomowy kopiec można umieścić w standardowej tablicy, odwzorowując każdą pozycję z kopca binarnego na unikatowy indeks. Każde pole z obramowaniem z kresek zawiera liczbę całkowitą, która odpowiada indeksowi z tablicy reprezentującemu dany element z kopca binarnego. Także tu w kopcu binarnym pokazuję tylko priorytety elementów.

Każdy element jest powiązany z pozycją w tablicy `storage[]`. Aby uprościć obliczenia, pozycja `storage[0]` nie jest używana i nigdy nie zawiera elementu. Pierwszy element, o priorytecie 15, jest umieszczany na pozycji `storage[1]`. Widać tu, że lewe dziecko, o priorytecie 13, jest umieszczane na pozycji `storage[2]`. Jeśli element z pozycji `storage[k]` ma lewe dziecko, znajduje się ono na pozycji `storage[2*k]`. Rysunek 4.17 potwierdza tę obserwację (wystarczy przyjrzeć się polom z obramowaniem z kresek). Podobnie, jeśli element z pozycji `storage[k]` ma prawe dziecko, znajduje się ono na pozycji `storage[2*k+1]`.



Rysunek 4.17. Zapisywanie kopca binarnego typu max w tablicy

Dla `k > 1` rodzic elementu z pozycji `storage[k]` znajduje się na pozycji `storage[k//2]`. Wartość `k//2` to liczba całkowita uzyskana przez zaokrąglenie w dół wyniku dzielenia `k` przez 2. Umieszczenie pierwszego elementu z kopca na pozycji `storage[1]` pozwala wykonać dzielenie całkowitoliczbowe w celu obliczenia lokalizacji rodzica danego elementu. Na przykład rodzic elementu z pozycji `storage[5]` (o priorytecie 11) znajduje się na pozycji `storage[2]`, ponieważ 5//2 = 2.

Element z pozycji `storage[k]` jest poprawny, jeśli $0 < k \leq N$, gdzie N to liczba elementów w danym kopcu binarnym typu max. To oznacza, że element z pozycji `storage[k]` nie ma dzieci, jeśli $2 \cdot k > N$. Na przykład element z pozycji `storage[10]` (o priorytecie 1) nie ma lewego dziecka, ponieważ $2 \cdot 10 = 20 > 18$. Wiadomo też, że element z pozycji `storage[9]` (który przez zbieg okoliczności ma priorytet 9) nie ma prawego dziecka, ponieważ $2 \cdot 9 + 1 = 19 > 18$.

Implementacja „wypływania" i „zatapiania"

Proces tworzenia kopca binarnego typu max zacznij od utworzenia klasy `Entry`, która obejmuje pole z wartością (`value`) i pole z powiązanym priorytetem (`priority`):

```
class Entry:
  def __init__(self, v, p):
    self.value = v
    self.priority = p
```

Kod z listingu 4.2 zapisuje kopiec binarny typu max w tablicy `storage`. W momencie tworzenia obiektu długość tablicy `storage` jest o 1 większa niż parametr `size`, aby można było przeprowadzić opisane wcześniej obliczenia, w których pierwszy element jest zapisywany na pozycji `storage[1]`.

Używane są tu dwie metody pomocnicze, które upraszczają omawianie kodu. Wielokrotnie widziałeś już, jak sprawdzałem, czy jeden z elementów ma większy priorytet niż inny element. Funkcja `less(i, j)` zwraca wartość `True`, gdy priorytet elementu z pozycji `storage[i]` jest mniejszy od priorytetu elementu z pozycji `storage[j]`. Ponadto w trakcie „wypływania” lub „zatapiania” elementów trzeba przestawiać dwa elementy. Funkcja `swap(i, j)` przestawia elementy z pozycji `storage[i]` i `storage[j]`.

Listing 4.2. Implementacja kopca z funkcjami enqueue() i swim()

```
class PQ:
  def less(self, i, j):                ❶
    return self.storage[i].priority < self.storage[j].priority

  def swap(self, i, j):                ❷
    self.storage[i],self.storage[j] = self.storage[j],self.storage[i]

  def __init__(self, size):            ❸
    self.size = size
    self.storage = [None] * (size+1)
    self.N = 0

  def enqueue(self, v, p):             ❹
    if self.N == self.size:
      raise RuntimeError ('Kolejka priorytetowa jest pełna!')

    self.N += 1
    self.storage[self.N] = Entry(v, p)
    self.swim(self.N)

  def swim(self, child):                             ❺
    while child > 1 and self.less(child//2, child): ❻
      self.swap(child, child//2)                     ❼
      child = child//2                               ❽
```

❶ Funkcja `less()` określa, czy element z pozycji `storage[i]` ma priorytet niższy niż element z pozycji `storage[j]`.

❷ Funkcja `swap()` przestawia elementy z pozycji `i` oraz `j`.

❸ Pozycje od `storage[1]` do `storage[size]` przechowują elementy. Pozycja `storage[0]` nie jest używana.

❹ Aby dodać do kolejki element (`v`, `p`), należy umieścić go w następnej pustej lokalizacji i pozwolić mu „wypłynąć” w górę.

❺ Funkcja `swim()` modyfikuje tablicę `storage`, aby była zgodna z właściwością uporządkowania kopca.

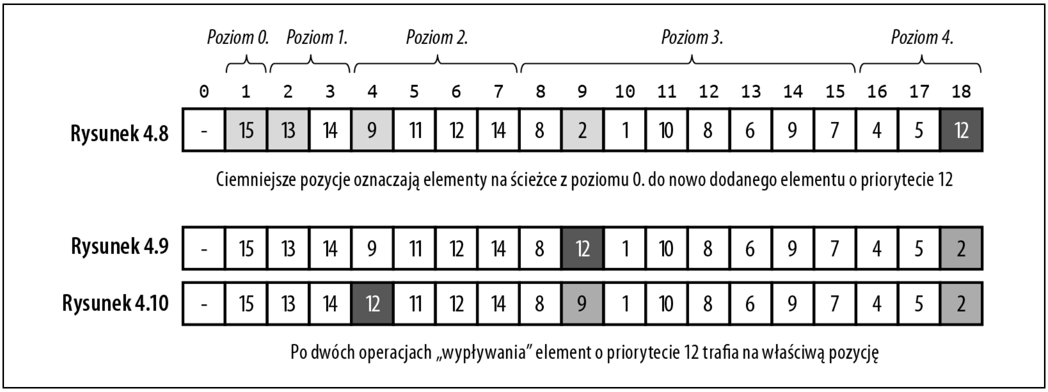
❻ Rodzic elementu z pozycji `storage[child]` znajduje się na pozycji `storage[child//2]`, gdzie `child//2` to liczba całkowita uzyskana przez zaokrąglenie w dół wyniku dzielenia `child` przez 2.

❼ Przestawianie elementu z pozycji `storage[child]` z jego rodzicem z pozycji `storage[child//2]`.

❽ Przechodzenie na następny poziom po przestawianiu elementu z pozycji `child` z jego rodzicem, gdy jest to konieczne.

Metoda `swim()` jest bardzo krótka. Element z pozycji `child` to element świeżo dodany do kolejki, natomiast `child//2` to jego rodzic (jeśli taki istnieje). Jeżeli rodzic ma niższy priorytet niż dziecko, elementy należy przestawić, a cały proces jest kontynuowany na następnych poziomach.

Na rysunku 4.18 pokazane są zmiany w tablicy `storage` wywołane przez operację `enqueue(wartość, 12)` w kopcu z rysunku 4.8. Każdy kolejny wiersz odpowiada jednemu z wcześniejszych rysunków i przedstawia elementy przestawione w tablicy `storage`. Ostatni wiersz reprezentuje kopiec binarny typu max spełniający właściwości struktury i uporządkowania.



Rysunek 4.18. Zmiany w tablicy storage po dodaniu elementu do kopca z rysunku 4.8

Ścieżka od elementu z najwyższego poziomu do nowo wstawionego elementu o priorytecie 12 obejmuje 5 elementów wyróżnionych kolorem na rysunku 4.18. Po 2 powtórzeniach pętli `while` z funkcji `swim()` element o priorytecie 12 jest przestawiany z rodzicem i ostatecznie trafia na pozycję `storage[4]`; właściwość uporządkowania kopca jest wtedy spełniona. Liczba operacji przestawiania nigdy nie jest większa niż `log`(N), gdzie N to liczba elementów w kopcu binarnym.

Kod z listingu 4.3 obejmuje metodę `sink()`, która naprawia strukturę kopca binarnego typu max po wywołaniu metody `dequeue()`.

Listing 4.3. Implementacja kopca uzupełniona o metody dequeue() i sink()

```
def dequeue(self):
  if self.N == 0:
    raise RuntimeError ('Kolejka priorytetowa jest pusta!')

  max_entry = self.storage[1]             ❶
  self.storage[1] = self.storage[self.N]  ❷
  self.storage[self.N] = None
  self.N -= 1                             ❸
  self.sink(1)
  return max_entry.value                  ❹

def sink(self, parent):
  while 2*parent <= self.N:               ❺
    child = 2*parent
    if child < self.N and self.less(child, child+1): ❻
      child += 1
    if not self.less(parent, child)        ❼
      break
    self.swap(child, parent)               ❽
    parent = child
```

❶ Zapisanie elementu o najwyższym priorytecie z poziomu 0.

❷ Zastąpienie elementu z pozycji `storage[1]` elementem z dolnego poziomu kopca. Usunięcie tego ostatniego elementu z pierwotnej lokalizacji z tablicy `storage`.

❸ Zmniejszenie liczby elementów *przed* wywołaniem metody `sink` dla elementu z pozycji `storage[1]`.

❹ Zwrócenie wartości powiązanej z elementem o najwyższym priorytecie.

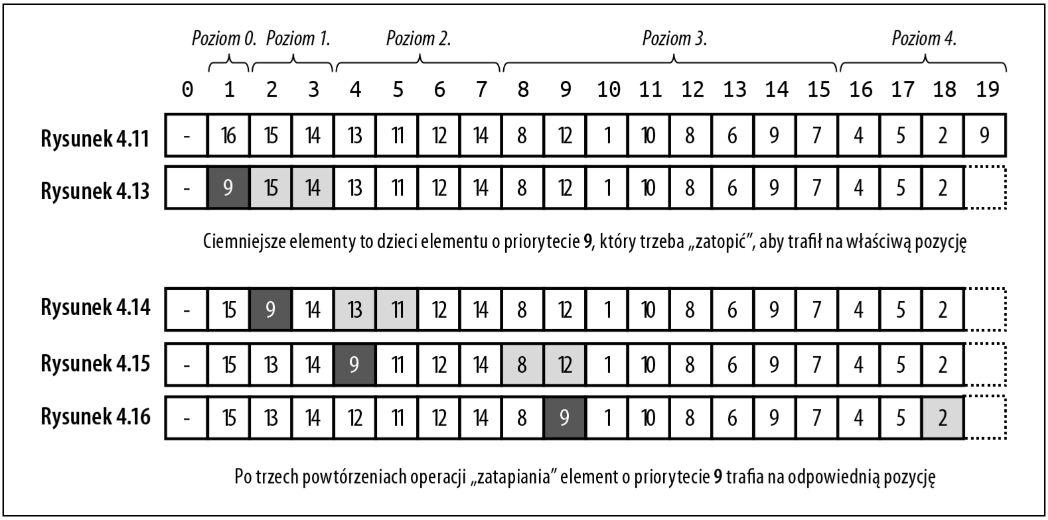
❺ Kontynuowanie sprawdzania tak długo, jak długo rodzic ma dziecko.

❻ Wybranie prawego dziecka, jeśli istnieje i jest większe niż lewe dziecko.

❼ Jeśli rodzic (`parent`) *nie* jest mniejszy niż dziecko (`child`), właściwość uporządkowania kopca jest spełniona.

❽ Przestawianie elementów, jeśli jest to konieczne, i dalsze „zatapianie" elementu z dzieckiem (`child`) jako nowym rodzicem (`parent`).

Na rysunku 4.19 pokazane są zmiany w tablicy `storage` spowodowane przez wywołanie `dequeue()` dla początkowego kopca binarnego typu max z rysunku 4.11. Pierwszy wiersz na rysunku 4.19 przedstawia tablicę z 19 elementami. W drugim wierszu ostatni element z kopca (o priorytecie 9) jest przestawiany i trafia na pozycję pierwszego elementu kopca binarnego typu max, co narusza właściwość uporządkowania kopca. Ponadto kopiec zawiera teraz tylko 18 elementów, ponieważ jeden element został usunięty.



Rysunek 4.19. Zmiany w tablicy storage po wywołaniu metody dequeue() dla kopca z rysunku 4.11

Po trzech kolejnych iteracjach pętli `while` w metodzie `sink()` element o priorytecie 9 trafia na właściwą pozycję i właściwość uporządkowania kopca jest spełniona. W każdym wierszu pierwszym wyróżnionym elementem jest ten o priorytecie 9, a elementy z ciemnym tłem po prawej stronie to jego dzieci. Gdy priorytet rodzica (9) jest mniejszy niż przynajmniej jednego z dzieci, rodzica trzeba „zatopić", przestawiając z dzieckiem o wyższym priorytecie. Liczba przestawień nigdy nie jest większa niż `log`(N).

Metoda `sink()` jest trudna do zwizualizowania, ponieważ (inaczej niż w metodzie `swim()`) nie występuje tu prosta ścieżka. W końcowej reprezentacji tablicy `storage` na rysunku 4.19 widać, że wyróżniony element o priorytecie 9 ma tylko jedno dziecko (wyróżnione ciemniejszym tłem; priorytet 2). Po zakończeniu wykonywania metody `sink()` wiadomo, że „zatapiany" element albo dotarł na pozycję `p` i nie ma dzieci (ponieważ $2 \cdot p$ jest nieprawidłowym indeksem w tablicy `storage`, bo jest większe niż N), albo ma priorytet większy lub równy (nie mniejszy) względem większego z dzieci.

> Kolejność wykonywania instrukcji w metodzie `dequeue()` jest *krytyczna*. Przede wszystkim musisz zmniejszyć N o 1 przed wywołaniem `sink(1)`. W przeciwnym razie metoda `sink()` mylnie przyjmie, że indeks z tablicy `storage` odpowiadający elementowi właśnie usuniętemu z kolejki *nadal jest częścią kopca*. W kodzie widać, że do `storage[N]` przypisywana jest wartość `None`; dzięki temu usunięty element nie zostanie błędnie uznany za część kopca.

Jeśli chcesz się przekonać, że kod metody `dequeue()` jest poprawny, rozważ jego działanie dla kopca zawierającego tylko jeden element. Metoda `dequeue()` pobierze `max_entry` i przypisze 0 do N przed wywołaniem metody `sink()`, która nie wykona żadnych operacji, ponieważ $2 \cdot 1 > 0$.

Podsumowanie

Kopiec binarny pozwala opracować wydajną implementację kolejki priorytetowej. Istnieje wiele algorytmów, opisanych na przykład w rozdziale 7., które bazują na kolejkach priorytetowych.

- Dodawanie do kolejki elementu (wartość, priorytet) (metoda `enqueue()`) ma złożoność O(`log` N).
- Usuwanie z kolejki elementu o najwyższym priorytecie (metoda `dequeue()`) ma złożoność O(`log` N).
- Liczbę elementów w kopcu można podać w czasie O(1).

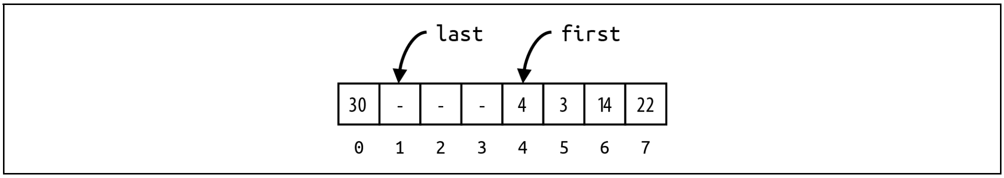
W tym rozdziale skupiłem się wyłącznie na kopcach binarnych typu max. Wystarczy wprowadzić jedną drobną zmianę, aby uzyskać *kopiec binarny typu min*, gdzie bardziej priorytetowe elementy mają mniejsze wartości priorytetu. Takie kopce będą potrzebne w rozdziale 7. Na listingu 4.2 wystarczy zmodyfikować metodę `less()` i użyć w niej operatora większości (>) zamiast mniejszości (<). Reszta kodu pozostaje taka sama.

```
def less(self, i, j):
  return self.storage[i].priority > self.storage[j].priority
```

Choć kolejka priorytetowa może stawać się większa lub mniejsza, w implementacji bazującej na kopcu określona jest początkowa wielkość M, co pozwala zapisać N < M elementów. Gdy kopiec jest pełny, do kolejki priorytetowej nie można dodać kolejnych elementów. Można jednak automatycznie zwiększać (i zmniejszać) tablicę na dane, podobnie jak w rozdziale 3. Dopóki stosujesz geometryczną zmianę rozmiaru, która polega na podwajaniu rozmiaru tablicy na dane po jej zapełnieniu, ogólny zamortyzowany koszt wykonywania metody `enqueue()` będzie równy O(`log` N).

Ćwiczenia

1. Można użyć tablicy `storage` o stałej wielkości do wydajnego zaimplementowania kolejki, tak aby operacje `enqueue()` i `dequeue()` miały złożoność O(1). Ta technika jest nazywana *kolejką cykliczną* i zastosowane jest w niej nowatorskie podejście polegające na tym, że pierwsza wartość w tablicy nie zawsze jest zapisana na pozycji `storage[0]`. Zamiast tego należy aktualizować pola `first` (indeks najstarszej wartości w kolejce) i `last` (indeks, pod którym należy umieścić w kolejce następną dodawaną wartość), jak ilustruje to rysunek 4.20.



Rysunek 4.20. Używanie tablicy jako kolejki cyklicznej

Gdy dodajesz wartości do kolejki i je pobierasz, musisz starannie modyfikować wspomniane pola. Warto śledzić wartość N, czyli liczbę elementów już zapisanych w kolejce. Czy potrafisz dokończyć implementację z listingu 4.4 i zapewnić, że operacje będą wykonywane w stałym czasie? W kodzie użyj operatora modulo, `%`.

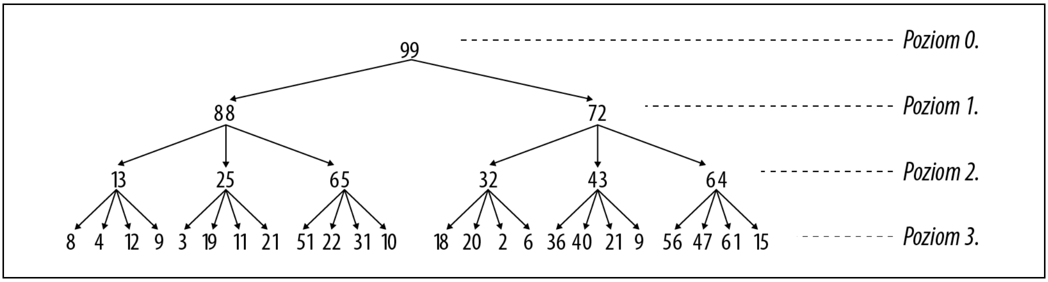
Listing 4.4. Dokończ implementację kolejki cyklicznej w klasie Queue

```
class Queue:
  def __init__(self, size):
    self.size = size
    self.storage = [None] * size
    self.first = 0
    self.last = 0
    self.N = 0

  def is_empty(self):
    return self.N == 0

  def is_full(self):
    return self.N == self.size

  def enqueue(self, item):
    """Jeśli kolejka nie jest pełna, metoda enqueue ma złożoność O(1)."""

  def dequeue(self):
    """ Jeśli kolejka nie jest pełna, metoda dequeue ma złożoność O(1)."""
```

2. Wstaw $N = 2^k - 1$ elementów rosnąco do pustego kopca binarnego typu max o wielkości N. Czy na podstawie analizy uzyskanej tablicy (z pominięciem nieużywanego indeksu 0) potrafisz przewidzieć indeksy największych `k` wartości w tablicy na dane? Czy jeśli wstawisz N elementów malejąco do pustego kopca binarnego typu max, to czy możesz przewidzieć indeksy *wszystkich* N wartości?

3. Dane są dwa kopce binarne typu max o wielkościach M i N. Zaprojektuj algorytm, który zwraca tablicę o wielkości M + N, zawierającą uporządkowane rosnąco elementy z obu kopców. Algorytm ma działać w czasie O(M `log` M + N `log` N). Wygeneruj tabelę z czasami wykonywania algorytmu, aby uzyskać empiryczny dowód na jego działanie.

4. Użyj kopca binarnego typu max, aby znaleźć `k` najmniejszych wartości w kolekcji N elementów w czasie O(N `log k`). Wygeneruj tabelę z czasami wykonywania algorytmu, aby uzyskać empiryczny dowód na jego działanie.
5. W kopcu binarnym typu max każdy rodzic ma dwoje dzieci. Rozważ inną strategię, którą nazwałem tu *kopiec silnia*; pierwszy element ma w niej dwoje dzieci, każde z tych dzieci ma troje dzieci (wnuków), każdy z wnuków ma czworo dzieci itd., jak ilustruje to rysunek 4.21. Na każdym kolejnym poziomie elementy mają o jedno dziecko więcej. Nadal obowiązują tu właściwości struktury i uporządkowania kopca. Opracuj implementację, zapisując kopiec silnia w tablicy. Udowodnij empirycznie, że wydajność takiej struktury jest niższa niż kopca binarnego typu max. Klasyfikacja złożoności czasowej jest tu bardziej skomplikowana, ale powinno udać Ci się stwierdzić, że złożoność wynosi tu O(`log` N/`log`(`log` N)).



Rysunek 4.21. Nowa struktura — kopiec silnia

6. Użyj strategii geometrycznego zwiększania rozmiaru z rozdziału 3., aby rozbudować implementację `PQ` z tego rozdziału i automatycznie zmieniać wielkość tablicy na dane, podwajając ją, gdy jest pełna, i zmniejszając o połowę, gdy jest zapełniona w ¼.
7. Iterator kopca bazującego na tablicy powinien zwracać wartości *w kolejności, w jakiej byłyby pobierane z kolejki*, ale bez modyfikowania samej tablicy (ponieważ iterator nie powinien mieć efektów ubocznych). Nie da się jednak łatwo uzyskać takiego efektu, ponieważ pobieranie wartości z kolejki zmienia strukturę kopca. Jednym z rozwiązań jest utworzenie funkcji generatora `iterator(pq)`, która przyjmuje kolejkę priorytetową, `pq`, i tworzy odrębną kolejkę priorytetową `pqit`, której wartościami są indeksy w tablicy na dane (`storage`) kolejki `pq`, a priorytety są równe priorytetom powiązanych wartości. Kolejka `pqit` powinna bezpośrednio używać tablicy `storage` kolejki `pq` do pobierania zwracanych elementów, ale bez modyfikowania zawartości tej tablicy.

   Uzupełnij pokazaną poniżej implementację, która zaczyna od wstawienia do kolejki `pqit` indeksu 1, odpowiadającego elementowi o najwyższym priorytecie z `pq`. Dokończ pętlę `while`:

   ```
   def iterator(pq):
     pqit = PQ(len(pq))
     pqit.enqueue(1, pq.storage[1].priority)

     while pqit:
       idx = pqit.dequeue()
       yield (pq.storage[idx].value, pq.storage[idx].priority)
       ...
   ```

   Dopóki pierwotna kolejka `pq` nie zostanie zmodyfikowana, iterator będzie zwracał każdą wartość zgodnie z priorytetami.

Skorowidz

T

W

Z


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Algorytmy:
oto, co najważniejsze w erze informacji!

Doskonałe opanowanie dowolnego języka programowania nie wystarczy do tego, aby stać się świetnym programistą czy deweloperem. Konieczne jest również zdobycie praktycznej wiedzy dotyczącej algorytmów. Oznacza to, że aby pisać lepszy kod, podczas rozwiązywania rzeczywistych problemów trzeba umieć korzystać z algorytmów, włączając w to ich budowanie, modyfikację i implementację. Niezależnie od tego, jaką dziedziną informatyki się zajmujesz, biegłość w posługiwaniu się algorytmami w wymierny sposób ułatwi Ci pracę i poprawi jej rezultaty.

Ta książka jest przystępnym wprowadzeniem do wiedzy o algorytmach wraz z przykładami implementacji napisanymi w Pythonie. Oprócz praktycznego omówienia algorytmów znalazło się tu wyjaśnienie takich pojęć jak klasy złożoności czy analiza asymptotyczna. Dokładnie omówiono także najważniejsze algorytmy, w tym różne sposoby haszowania, sortowania czy przeszukiwania. Tam, gdzie to niezbędne, wprowadzono struktury danych języka Python. Z poradnika programiści i testerzy dowiedzą się, w jaki sposób wykorzystywać algorytmy do pomysłowego rozwiązywania problemów obliczeniowych. Zrozumienie treści ułatwiają ciekawe materiały wizualne i ćwiczenia utrwalające, które pozwolą na przetestowanie zdobytej wiedzy w praktyce.

W książce między innymi:

- podstawowe algorytmy wykorzystywane w inżynierii oprogramowania
- standardowe strategie wydajnego rozwiązywania problemów
- cena złożoności czasowej kodu z wykorzystaniem notacji dużego O
- praktyczne stosowanie algorytmów z wykorzystaniem bibliotek i struktury danych Pythona
- główne zasady działania ważnych algorytmów

George Heineman jest naukowcem i wykładowcą akademickim. Od ponad 20 lat zajmuje się inżynierią oprogramowania i algorytmiką. Jest autorem i współautorem książek technicznych, często też prowadzi szkolenia dotyczące stosowania algorytmów. Ma nietypową pasję: łamigłówki. Jest twórcą odmiany sudoku Sujiken® i Trexagon.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-8799-7
9 788328 387997
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 59,00 zł