Zadanie 1.
Pokaż w jaki sposób można efektywnie przetrzymywać kopiec binarny rozmiaru n w tablicy długości n. Jak to wygląda dla kopca d-arnego?

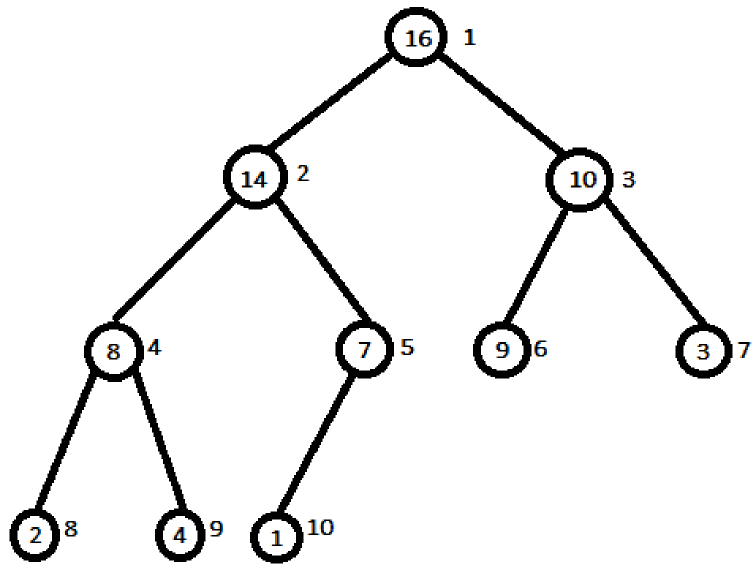
Kopiec binarny może być rozpatrywany jako drzewo binarne oraz jako tablica. Weźmy przykładowy kopiec typu max(tzn. ojciec ma większą wartość niż synowie). Jego reprezentacja w postaci drzewa binarnego wygląda tak, że każdy węzeł ma wartość oraz indeks.

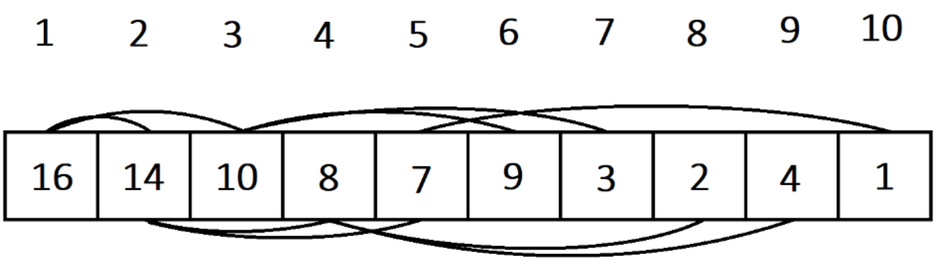
Możemy to przełożyć na tablicę o rozmiarze $1 + 2^1 + 2^2 + \dots + 2^{n-1}$ + węzły najniższego poziomu. Takie odwzorowanie ma sens, bo łatwo można odnaleźć indeks ojca, lewego syna oraz prawego syna pewnego węzła x. Mając dany indeks *i* węzła x:

Szukanie indeksu ojca x: *floor*(*i*/2)

Szukanie indeksu lewego syna x: 2*i*

Szukanie indeksu prawego syna x: 2*i* + 1





Aby upewnić się, że struktura może być przetrzymywana efektywnie, należy jeszcze rozważyć kwestię operacji, jakich można używać na kopcu binarnym. Do umożliwienia wykonania tych operacji dodamy naszemu kopcowi atrybut *size*, który ułatwi znaleźć ostatni element. Elementy będą przechowywane na indeksach 1, 2, 3, …, *size.* Podstawowe działania na kopcach:

Insert – element dodawany jest na koniec tablicy(atrybut *size* jest inkrementowany), a następnie przywracane są właściwości kopca w O(log n). Następuje to poprzez porównywanie dodanego elementu ‘x’ z początkowo jego ojcem i jeśli istnieje nieporządek względem nich, to ‘x’ jest porównywany z ojcem jego ojca, aż trafi na odpowiednie miejsce.

(ExtractMax)

Delete – element jest usuwany z początku tablicy, ostatni element przestawiamy na pierwszy indeks(atrybut *size* jest dekrementowany), a następnie wykonujemy procedurę Heapify, by przy przywrócić własności kopca. Delete ma złożoność wywoływanej przez siebie operacji Heapify.

Heapify – służy do utrzymania własności kopca. Przekazany element ‘x’ jest porównywany z obydwoma synami, aby ustalić największy element. Jeśli największy element jest synem ‘x’ to jest z nim zamieniany. Następnie, jeśli ‘x’ nie jest liściem i był w poprzednim kroku zamieniany, to wykonujemy Heapify rekurencyjnie dla ‘x’. Czas działania Heapify może być opisany rekurencją

$T(n) \leq T(2n/3) + \Theta(1)$, z czego wynika, że ma złożoność $O(\log n)$.

IncreaseKey – służy do zwiększenia wartości klucza elementu. W tym celu modyfikujemy wartość klucza podanego ‘x’, a następnie podobnie jak w operacji insert porównujemy ‘x’ z jego ojcem, aby ustalić porządek. Porównywanie następuje dopóki ‘x’ jest większe od ojca. Podobnie jak w Insert złożoność IncreaseKey jest O(log n).

Złożoność operacji działających na indeksach tablicy nie różni się od złożoności operacji działających na drzewach, więc oznacza to, że możemy efektywnie przetrzymywać strukturę kopca w tablicy.

Przypadek kopca d-arnego jest bardzo podobny. Różnica pomiędzy kopcem binarnym i d-arnym jest taka, że kopiec binarny może mieć 2 synów, a kopiec d-arny może mieć ich d. Z tego powodu tablica kopca d-arnego ma rozmiar $1 + d^1 + d^2 + \ldots + d^{n-1}$ + węzły najniższego poziomu. Mając dany indeks i węzła x:

Szukanie indeksu ojca x: $\mathit{floor}((i+d-2)/d)$

Szukanie indeksu pierwszego syna x: $di-(d-2)$

Szukanie indeksu drugiego syna x: $di-(d-2) + 1$

…

Szukanie indeksu k-tego syna x: $di-(d-2) + (k-1)$

Można łatwo udowodnić, że powyższe rozumowanie jest prawdziwe szukając indeks rodzica k-tego syna pewnego węzła o indeksie i:

-1

$$\mathit{floor}(((di-(d-2) + (k-1)) +d-2)/d) = \mathit{floor}((di-d+2+k+d-2)/d) =$$

$$= \mathit{floor}((di+(k-1))/d) = \mathit{floor}(i + (k-1)/d) = i$$

Ostatni krok działa, ponieważ $(k-1) < d$

Przykład graficzny dla d = 3