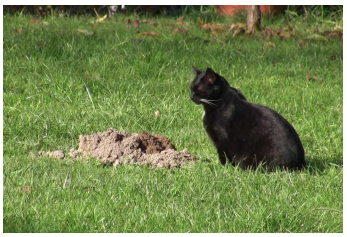
Kopiec binarny



`#własność_kopca` `#heapsort`
`#vector.back()` `#vector.pop_back()`

Czas na specyficzną strukturę danych (kontener), która zapewnia następującą funkcjonalność:

- odczytanie wartości największego elementu (w czasie stałym, czyli w czasie błysku ciupagi),
- usunięcie tegoż największego elementu (w czasie logarytmicznym),
- dodanie nowego elementu (również w czasie logarytmicznym).

Narzuca się pomysł, aby zrealizować taką strukturę przy pomocy posortowanej tablicy lub wektora. OK, o ile początkowe dwa punkty dałoby się zrealizować, to z trzecim będziemy mieć zgryz. Jak tu wstawić nowy element w środek posortowanego wektora? Trzeba by znaleźć odpowiednie miejsce: to da się zrobić przy pomocy *binsearch*-a w czasie logarytmicznym, ale jeśli coś chcemy wstawić do środka liniowej struktury, to trzeba by przesunąć cały kawał danych w prawo (lub w lewo – w zależności od sposobu posortowania), żeby zrobić miejsce dla nowego elementu.

Jeśli z kolei posłużylibyśmy się strukturą typu lista czy `deque`, wtedy co prawda wstawianie i usuwanie odbywałoby się w czasie stałym (miodzio!), natomiast wyszukiwanie właściwego miejsca do wstawienia zajmowałoby czas liniowy (kicha!).

Trzeba zatem wymyślić coś innego i na pewno nie będzie to prosta struktura liniowa. Zaproponujemy coś w rodzaju drzewa o narzuconych szczególnych właściwościach.

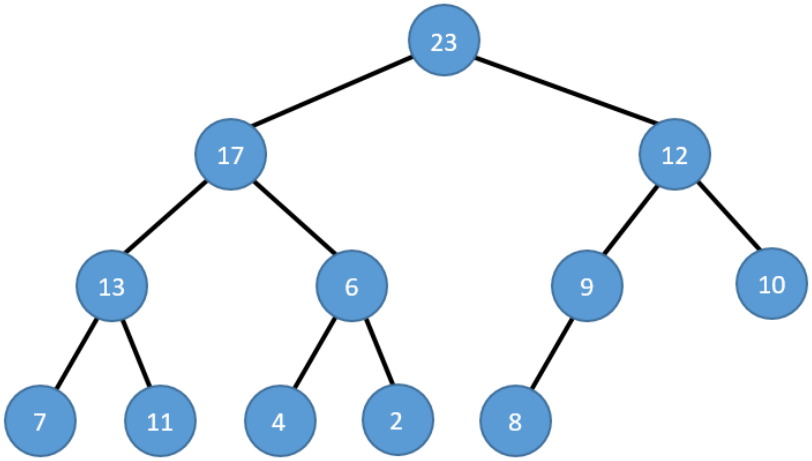
Drzewo binarne

Generalnie drzewem binarnym (ang. **binary tree**, wymawiaj: *bajnary tri*, z akcentem na literę *a* i długim *i*) nazywamy takie drzewo, w którym z każdego wierzchołka wyrastają (umownie: w dół) co najwyżej dwie krawędzie. Korzeń tego drzewa (ang. **root**, wymawiaj: *rut*, z długim *u*). znajduje się na samej górze, ponadto drzewo nie zawiera cykli krawędzi* i jest spójne.†

Oczywiście trudno nazwać powyższą definicję w pełni ścisłą, ale liczymy tutaj na intuicję Czytelnika. Drzewom poświęcimy zresztą wiele uwagi w innych *Miaukotach*.

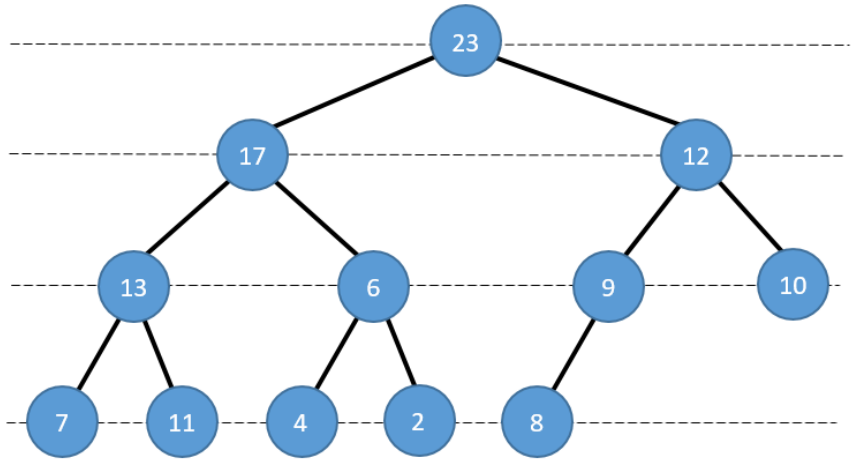
*Czyli nie da się biegać w kółko po krawędziach.
†Znaczy się jest w jednym kawałku.

Oto przykładowe drzewo binarne:



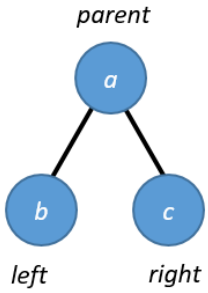
To szczególne drzewo ma jeszcze kilka ciekawych cech, ale o tym za chwilę. Na razie zauważamy korzeń drzewa – z wartością 23 – oraz jego *liście* z wartościami 7, 11, 4, 2, 8 oraz 10.

Przyjrzyjmy sie uważnie poziomom, na których znajdują się wierzchołki (węzły) drzewa:



Idąc poziomami od góry obserwujemy, że każdy z poziomów jest maksymalnie obsadzony (pełny) – z wyjątkiem najniższego, na którym wierzchołki są „dosunięte” do lewej strony.

Jest jeszcze coś: skupmy się na relacji pomiędzy trzema sąsiednimi węzłami (określanymi mianem **parent**, **left**, **right** – *rodzic*, *lewy*, *prawy*):



Węzły *lewy* oraz *prawy* będziemy określać wspólnym mianem *synów* lub *potomków*.

Zanotujmy, że dla każdej takiej trójki zachodzi relacja zwana *własnością kopca* (ang. **heap property**, wymawiaj: *hip property*, z długim *i* oraz akcentem na *o*):[‡]

$$b < a \qquad \text{oraz} \qquad c < a,$$

czyli

$$\mathit{left} < \mathit{parent} \qquad \text{oraz} \qquad \mathit{right} < \mathit{parent}.$$

I tak oto po raz pierwszy pojawiło się pojęcie *kopca binarnego* (ang. **binary heap**).[§] Istotnie, takim właśnie mianem określa się drzewo binarne spełniające opisane wyżej warunki, a na rysunku widzimy jego przykład. Teraz musimy tylko wymyślić sposób implementacji samej struktury danych oraz efektywnych funkcji realizujących zadania wymienione na początku tego podrozdziału.

Kopiec binarny jako tablica/wektor

Widząc drzewiastą strukturę kopca, można by powziąć podejrzenie, że jego implementacja opierać się będzie na wskaźnikach: z *parent*-a wiodą wskaźniki do *left*-a i *right*-a lub odwrotnie (lub też naraz). Oczywiście, jest to możliwe, ale w przypadku kopca binarnego dwie jego cechy pozwalają na znaczące uproszczenie implementacji:

- jest to drzewo binarne, czyli z każdego węzła wiodą krawędzie do co najwyżej dwóch potomków (czyli w dół),
- każdy poziom węzłów jest dopełniony – tylko najniższy poziom może być niepełny, a i wtedy wszystkie węzły na nim muszą być zgromadzone z jego lewej strony.

Takie ograniczenia pozwalają na efektywną (w sensie zajętości pamięci) reprezentację zawartości kopca przy użyciu struktury liniowej: tablicy lub wektora. Zapisujemy w niej od lewej kolejne poziomy węzłów, idąc od góry poziomami węzłów. Prezentowany wyżej kopiec zapiszemy w następujący sposób:

23	17	12	13	6	9	10	7	11	4	2	8
0	1	2	3	4	5	6	7	8	9	10	11

Musimy zapewnić odpowiednią komunikację miedzy komórkami tablicy, tak aby odpowiadała dokładnie zależnościom pomiędzy węzłami kopca. Zaczniemy od napisania funkcji `parent()`, która dla danej komórki oblicza położenie jej rodzica. Zobaczmy, komórki 1, 2 mają rodzica 0, komórki 3, 4 – rodzica 1, komórki 5, 6 – rodzica 2, i tak dalej. Na oko widać, że spełniona jest relacja:

$$\mathit{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor.$$

[‡]W naszym przykładzie wszystkie elementy struktury mają różne wartości (dla prostoty). W ogólniejszym przypadku trzeba użyć nieostrych nierówności.

[§]Istnieją inne rodzaje kopców, jak choćby kopiec dwumianowy czy kopiec Fibonacci'ego. Kopiec binarny jest najprostszym przykładem takiej struktury. Przy okazji warto wspomnieć, że relacja pomiędzy *b* oraz *c* (czyli pomiędzy *left* oraz *right* nie jest określona – nie wiadomo, co jest większe.

Dzielenie liczb całkowitych w języku C++ przebiega według takich właśnie reguł, zatem mamy:

```
int parent(int i)
{
  return (i - 1) / 2;
}
```

Teraz funkcja `left()`: dla argumentu 0 powinniśmy otrzymać wartość 1, dla 1 – wartość 3, dla 2 – wartość 5, i tak dalej. Jako żywo krystalizuje się wzór:

$$left(i) = 2i + 1,$$

czyli konkretnie:

```
int left(int i)
{
  return 2 * i + 1;
}
```

Funkcję `right()` łatwo napisać:

```
int right(int i)
{
  return 2 * i + 2;
}
```

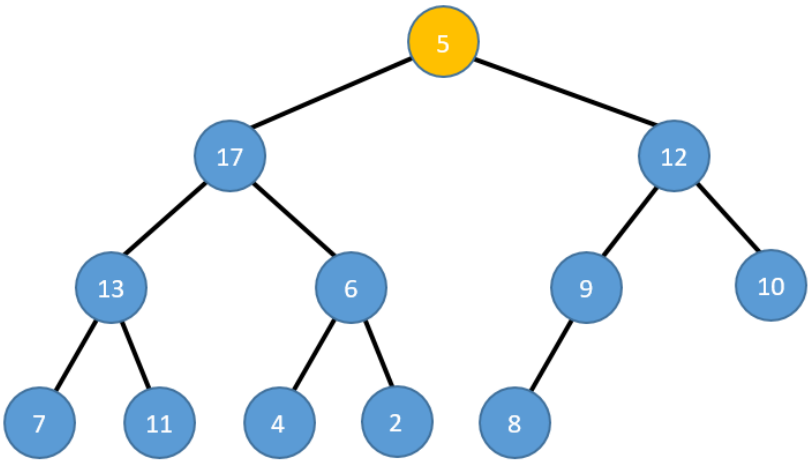
lub nawet tak:

```
int right(int i)
{
  return left(i) + 1;
}
```

Obecnie możemy już przystąpić do implementacji właściwych funkcji odpowiedzialnych za działanie kopca.

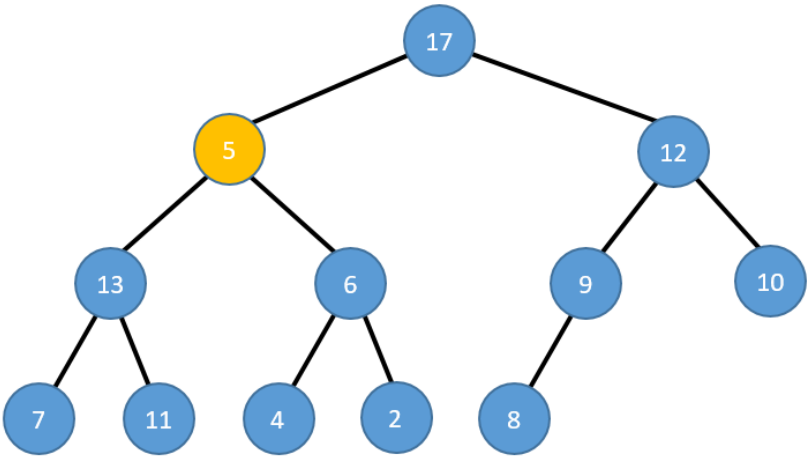
Funkcja `heapify()`

Nasz przykładowy kopiec jest zbudowany dokładnie według opisanych przez nas zasad. Załóżmy jednak, że w wyniku pewnego godnego pożałowania zajścia, na jego szczycie znalazł się jakiś element, który pasuje tam jak facet z kebabem do synagogi, na przykład:



Procedura `heapify()`[¶] (wymawiaj: *hipifaj*, z akcentem na *a*) ma za zadanie przywrócić własność kopca w całej strukturze, począwszy od wskazanego trefnego elementu. W naszym przypadku jest to ta nieszczęsna piątka na szczycie kopca.

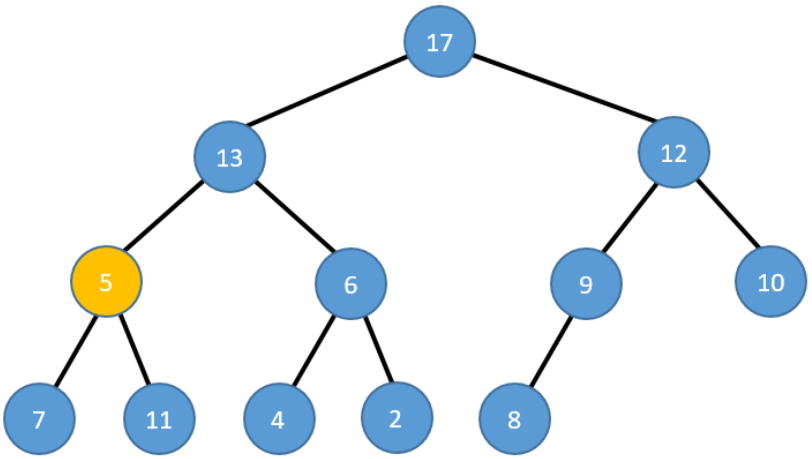
Patrzymy na ten wskazany element oraz na jego potomków i robimy między nimi zawody: który największy? W szranki stają elementy o wartościach 5, 17 i 12. Oczywiście wygrywa 17 i zamieniamy ten element z tą niewydarzoną piątką:



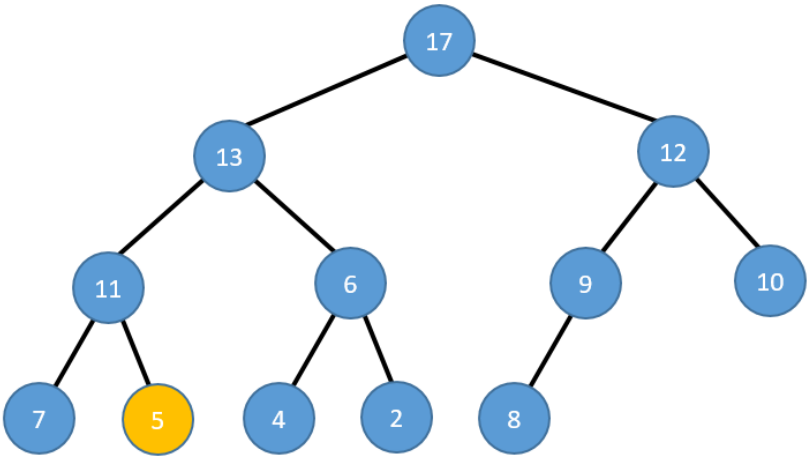
Piątka zeszła niżej, w lewą stronę, natomiast prawe poddrzewo (z korzeniem 12) nie uległo w ogóle zmianie – i tak już pozostanie. Wszelkie zmiany będą rozgrywać się poniżej elementu o wartości 5.

Teraz powtarzamy zawody w miejscu, gdzie obecnie znajduje się kebab, *pardon*, piątka. Z elementów 5, 13 i 6 wybieramy największy (13) i zamieniamy miejscami z piątką (poddrzewa o korzeniu w 6 nie tykamy):

[¶]Konia z rzędem temu, kto wymyśli sensowne polskie tłumaczenie tej nazwy – czasownik od słowa *kopiec*.



Dalej jeszcze jest coś nie tak i powtarzamy zawody, aż wreszcie piątka ląduje w bezpiecznym miejscu:



Ilość operacji (zamian miejsc sąsiadujących wierzchołków) jest pesymistycznie rzędu ilości poziomów kopca. Pesymistycznie, bo przecież może się zdarzyć że wędrówka kebabu (niech już tak zostanie) w dół skończy się wcześniej, niż na najniższym poziomie.[‖] A to znaczy, że funkcja `heapify()` ma złożoność logarytmiczną.

Czas na implementację tej funkcji: jej argumentem jest wektor reprezentujący kopiec (*heap*), jego rozmiar (*heap_size*) oraz numer węzła (i), od którego zaczynamy porządki. Od razu obliczamy indeks lewego i prawego potomka:

```
void heapify(vector<int> &heap, int heap_size, int i)
{
  int i_left = left(i), i_right = right(i);
  . . .
}
```

Rozmiar wektora podajemy jako oddzielną zmienną – przyda się nam to przy algorytmie sortowania Heapsort.

[‖] No i mogłaby się zacząć niżej, niż na samym szczycie kopca.

No i zaczynają się zawody: zmiennej i_max przypisujemy początkowo wartość i i zajmujemy się lewym potomkiem:

```
int i_max = i;
if(heap[i_left] > heap[i_max])
  i_max = i_left;
```

czyli teraz i_max wskazuje na większy z dwóch elementów: tego o indeksie i i tego o indeksie i_left (czyli $left(i)$).

Czyżby? Widzicie haczyk w powyższym kodzie? Otóż i_left wcale nie musi być poprawnym indeksem elementu kopca. Elementy drzewa, które są liśćmi, nie mają potomków, więc funkcja `left()` daje dla nich wynik wykraczający poza prawy koniec danych. Zatem w instrukcji warunkowej trzeba najpierw sprawdzić, czy i_left jest akceptowalne, a dopiero później ewentualnie porównywać wartości elementów:

```
int i_max = i;
if(i_left < heap_size && heap[i_left] > heap[i_max])
  i_max = i_left;
```

Kolejność porównań jest istotna: jeśli pierwszy warunek nie jest spełniony, wtedy drugi nie jest już w ogóle sprawdzany (koniunkcja!), czyli nie wykroczymy poza rozmiar struktury.

Podobne porównanie (a właściwie parę porównań) wykonujemy dla prawego potomka:

```
int i_max = i;
if(i_right < heap_size && heap[i_right] > heap[i_max])
  i_max = i_right;
```

No to teraz już na bank i_max jest indeksem zwycięzcy tych zawodów. Pozostaje sprawdzić, czy $i \neq i\_max$ i jeśli tak, wtedy zamieniamy miejscami $heap[i]$ oraz $heap[i\_max]$, a następnie... No właśnie, co następnie? Puszczamy funkcję `heapify()` dalej w dół (rekurencyjnie), aby znalazła miejsce dla elementu z pozycji i_max:

```
if(i != i_max)
{
  swap(heap[i], heap[i_max]);
  heapify(i_max);
}
```

No i tyle.

Dla porządku przedstawiamy pełną postać funkcji `heapify()`:

```
void heapify(vector<int> &heap, int heap_size, int i)
{
  int i_left = left(i), i_right = right(i);
  int i_max = i;
```

```
  if(i_left < heap_size && heap[i_left] > heap[i_max])
    i_max = i_left;
  int i_max = i;
  if(i_right < heap_size && heap[i_right] > heap[i_max])
    i_max = i_right;
  if(i != i_max)
  {
    swap(heap[i], heap[i_max]);
    heapify(heap, i_max);
  }
}
```

Funkcja ta będzie nam niezbędna do realizacji operacji usuwania największego elementu kopca.

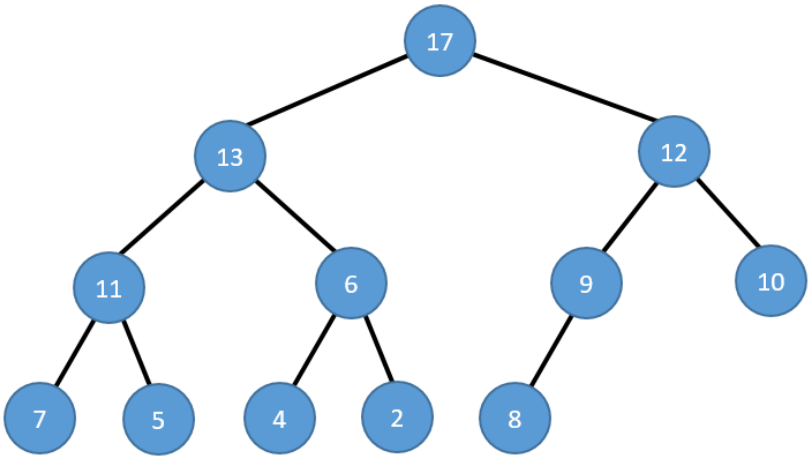
Największy element kopca

Samo określenie wartości największego elementu kopca (bezinwazyjne) to po prostu odczytanie elementu na szczycie stosu – jest to element wektora o indeksie 0. Zajmie się tym funkcja `heap_top()`:

```
int heap_top(vector<int> &heap, int heap_size)
{
  return heap[0];
}
```

W tej funkcji ten drugi argument nie jest potrzebny, ale podajemy go, aby odwołania do wektora *heap* wszedzie wygladały tak samo.

Dla kopca, który rozważaliśmy ostatnio:

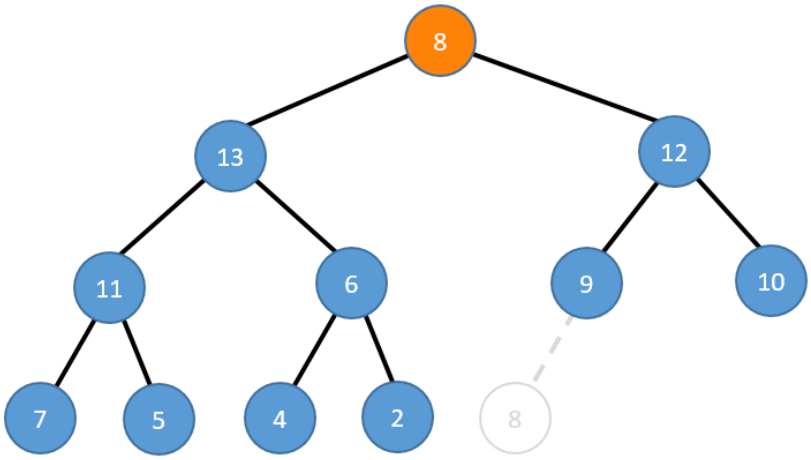


funkcja `heap_top()` zwróci wartość 17.

Z kolei usunięcie tego elementu wektora (funkcja `heap_pop()`) wiąże się z koniecznością zapewnienia (czy raczej: przywrócenia) własności kopca. Skoro element ze szczytu kopca musi zniknąć**, to czymś musimy utkać pozostałą po nim pustkę. Jedyny element, który możemy

**Jak starsza pani...

bezpiecznie ruszyć, to ostatni element kopca: na najniższym poziomie, po prawej stronie – dostęp do tego elementu jest na przykład przy pomocy funkcji `back()`. Ten właśnie element należy stamtąd zabrać i wstawić go na wolne miejsce szczycie wektora:

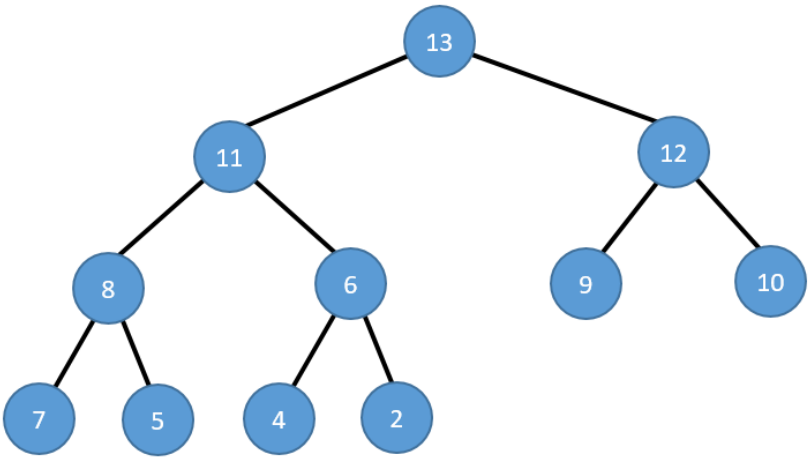


Ponieważ na pewno ten element pasować tam nie będzie (kebab!), więc należy uruchomić procedurę `heapify()`, aby zrobiła porządek (zauważmy symbol referencji przy argumencie *heap_size*):

```
void heap_pop(vector<int> &heap, int &heap_size)
{
  heap[0] = heap.back();
  heap.pop_back();
  heap_size--;
  heapify(heap, 0);
}
```

Funkcja `pop_back()` usuwa ostatni element wektora (dzieje się to w czasie stałym).

Otrzymujemy strukturę z przywróconą własnością kopca:



No i git.

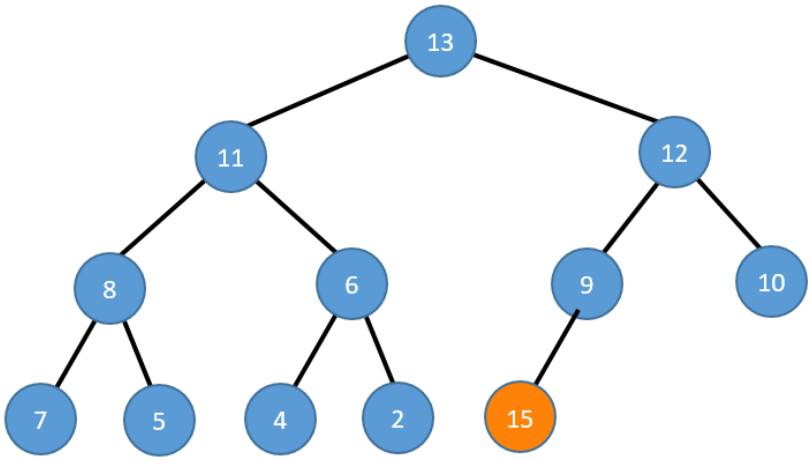
Dodawanie elementu do kopca

Dodając nowy element do kopca musimy w pierwszej kolejności zastanowić się, gdzie umieścić węzeł z nową wartością, a dopiero później bierzemy się za ewentualne porządkowanie kopca. Jedyne sensowne wolne miejsce znajduje się na samym dole.

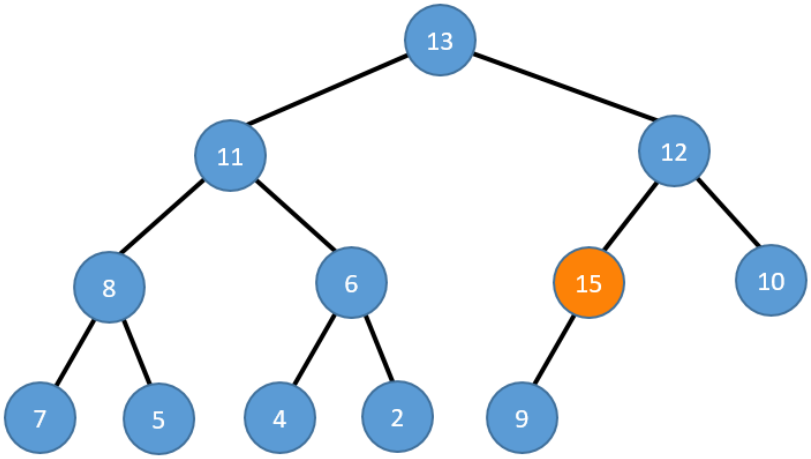
Mamy tutaj dwie możliwości:

- Najniższy poziom *nie jest* całkowicie wypełniony i wtedy nowy element ląduje na tym poziomie zaraz za ostatnim elementem,
- Najniższy poziom *jest* całkowicie wypełniony i nowy element zapoczątkowuje nowy poziom, całkiem z lewej strony.

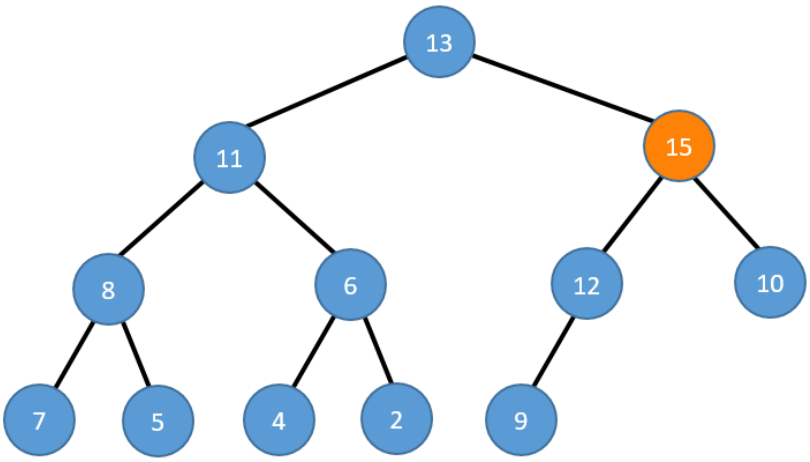
Tak czy owak, w wektorze reprezentującym kopiec wstawiamy ten element (tutaj o wartości 15) zaraz za jego ostatnim (dotychczas) elementem, zarazem zwiększając jego rozmiar o 1:



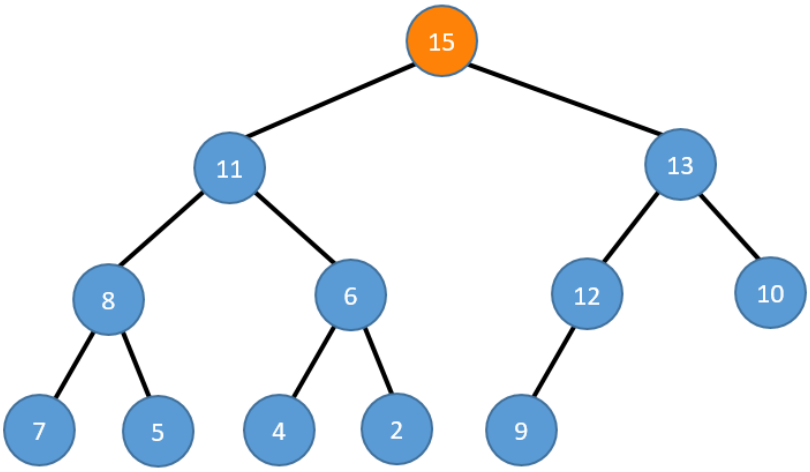
Teraz powinniśmy wykonać prace porządkowe idąc od dołu do góry, czyli w odwrotnym kierunku, niż w przypadku funkcji `heapify()`. Tym razem wystarczy porównanie tylko z *parent*-em: jeśli dany element (potomek) jest większy, wtedy zamieniamy węzły miejscami:



Tę operację musimy powtórzyć:



Dopiero po kolejnej zamianie miejsc element 15 jest umieszczony na właściwej pozycji:



Wędrówka nowego elementu od dołu do góry może zakończyć się z powodu dotarcia na sam szczyt kopca (jak w tym przypadku – rozpoznamy to po wartości indeksu 0) lub w razie napotkania *parent*-a, który jest większy od wstawianego elementu (lub jemu równy).

Pesymistyczna złożoność algorytmu wstawiania elementu do kopca jest taka sama, jak funkcji `heapify()`, czyli logarytmiczna.

Czeka nas jeszcze tylko implementacja funkcji `heap_insert()`. Ruch w kierunku *parent*-a jest wykonywany przy pomocy podstawienia:

```
i = parent(i);
```

Nowy element oznaczamy przez x. A oto i cała funkcja:

```
void heap_insert(vector<int> &heap, int &heap_size, int x)
{
  heap.push_back(x);
  heap_size++;
  int i = heap.size();
```

```
  while(i > 0 && heap[i] > heap[ parent(i) ])
  {
    swap(heap[i], heap[ parent(i) ]);
    i = parent(i);
  }
}
```

Zauważmy, że funkcjonalność kopca odpowiada dokładnie funkcjonalności kolejki priorytetowej, którą poznaliśmy już wcześniej. Rzeczywiście, biblioteczny adapter `priority_queue<>` jest zaimplementowany właśnie w oparciu o kopiec binarny (z wektorem pod spodem).

Algorytm Heapsort

Tam, gdzie mamy do czynienia z wyborem (za każdym razem) największego elementu, tam aż pachnie jakąś metodą sortowania. Tak jest również w przypadku kopca binarnego, którego funkcje (napisane przez nas w poprzednich sekcjach) mogą posłużyć do efektywnego posortowania wektora lub tablicy.

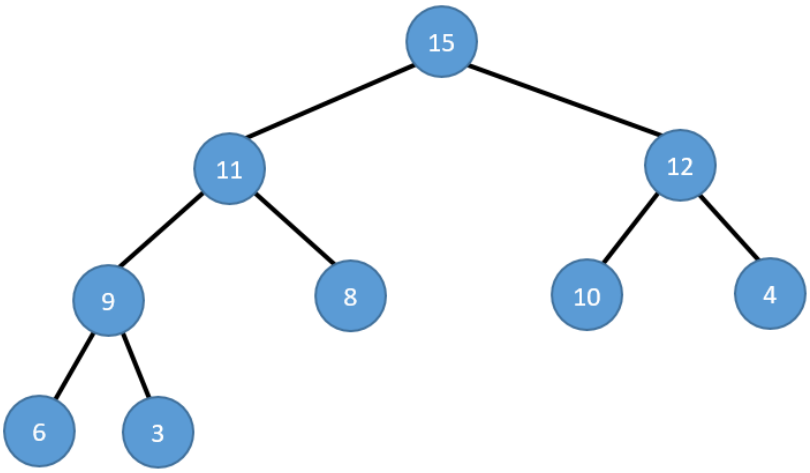
Załóżmy, że V jest wektorem liczb w losowej kolejności, na przykład:

```
vector<int> V = {12, 3, 15, 6, 8, 10, 4, 11, 9};
int V_size = V.size();
```

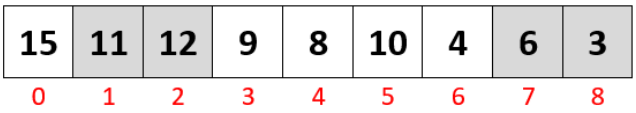
Najpierw zrobimy z niego pełnoprawny kopiec. Zauważmy, że wystarczy zająć się lewą połową tego wektora i poczochrać ją odpowiednio użytymi procedurami `heapify()`. Zaczniemy od pozycji $\lfloor V\_size/2 \rfloor$ i posuwać się w lewo, aż do pozycji 0:

```
for(int i = V_size / 2; i >= 0; i--)
  heapify(V, V_size, i);
```

Po takich karesach istotnie otrzymujemy poprawnie zbudowany kopiec:



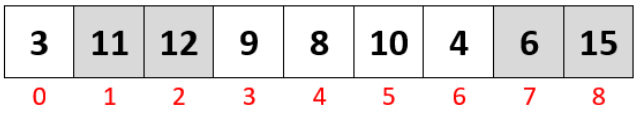
Zawartość wektora V przedstawia się następująco:



No, daleko temu jeszcze do posortowania, ale prosimy uzbroić się w cierpliwość. Skoro to jest kopiec, to największy element znajduje się na pewno na początku ($V[0]$). Możemy wyekspediować go na koniec wektora V (na pozycję $V\_size - 1$), a w to miejsce przenieść element z końca (taki myk robiliśmy w funkcji `heap_pop()`). Do tego możemy zmniejszyć wartość V_size, bo ostatni element jest już prawidłowo ustawiony:

```
swap(V[0], V[ V_size - 1 ]);
V_size--;
```

Teraz wektor V wygląda tak:



Na początku wektora mamy kebab, ale wystarczy puścić stamtąd `heapify()` i sprawy wrócą do normy:

```
heapify(V, V_size, 0);
```

Nic nie stoi na przeszkodzie, aby ponowić ten zestaw operacji – i ponawiać go do oporu, to znaczy, póki V_size jest przynajmniej równe 2:

```
while(V_size >= 2)
{
  swap(V[0], V[ V_size - 1 ]);
  V_size--;
  heapify(V, V_size, 0);
}
```

Przedstawimy teraz przykładowy program testujący tę metodę sortowania:

```
\include<bits/stdc++.h>
using namespace std;

// Tu należy wstawić definicje funkcji parent(), left(), right(),
// heapify(), heap_insert()
. . .

void heap_sort(vector<int> &V, int V_size)
{
  for(int i = V_size / 2; i >= 0; i--)
    heapify(V, V_size, i);
```

```
  while(V_size >= 2)
  {
    swap(V[0], V[ V_size - 1 ]);
    V_size--;
    heapify(V, V_size, 0);
  }
}

int main()
{
  vector<int> V = {12, 3, 15, 6, 8, 10, 4, 11, 9};
  int V_size = V.size();
  heap_sort(V, V_size);
  for(x : V)
    cout << x << ' ';
  cout << endl;
}
```

Zwracamy uwagę, aby w funkcji `heap_sort()` nie używać referencji przy V_size, aby nie zmienić jej wartości w funkcji `main()`.

A jaka jest złożoność tego algorytmu? Całkiem zacna, milordzie. Wystarczy przyjrzeć się pętli `while`: obchodzi ona około V_size razy i tyleż razy uruchamiana jest funkcja `heapify()` (jej złożoność jest logarytmiczna). Zbierając to razem dostajemy złożoność $N \lg_2 N$, gdzie N to V_size. Czyli standard europejski, może być.