ASD, Wyk. 06, 2022.03.31

1) Kolejki priorytetowe.
2) ich implementacja przez kopiec.
3) sortowanie przez kopcowanie.

Ad 1

W kolejce priorytetowej zadania z kolejki wyjmujemy zgodnie z ich priorytetem, a nie zgodnie z ich czasem zakolejkowania.

Czyli zadanie to para:

(z, p)

z – zadanie, p – priorytet

i mamy liniowy porządek na priorytetach.

Rozróżniamy min-kolejki priorytetowe

max- " " "

↳ min-kolejce szybciej wykonujemy zadania o mniejszych priorytetach.

Jak to zaimplementować?

Mamy dwie operacje: enqueue

dequeue

ale inna jest ich logika działania.

Co jeśli implementujemy kol. przy

użyciu tablicy.

Koszt pesymist. / Koszt średni	tablica uporządkowana	tablica nieuporządkowana
wyszukiwanie	n $\frac{n}{2}$	$O(1)$
usuwanie	$O(1)$	$O(n)$

Struktura kopca

pozwala zaimplementować

kolejkę priorytetową

z pesymistycznym czasem działania

$O(\log_2(n))$

dla obu operacji.

Co to jest kopiec?

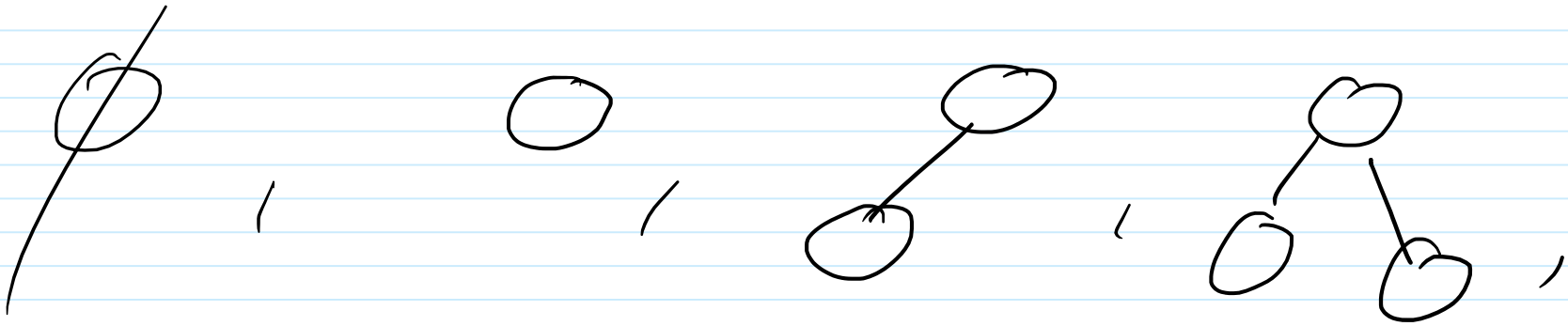
Zaimplementujemy max-kopiec w którym przechowujemy liczby całkowite (priorytety).

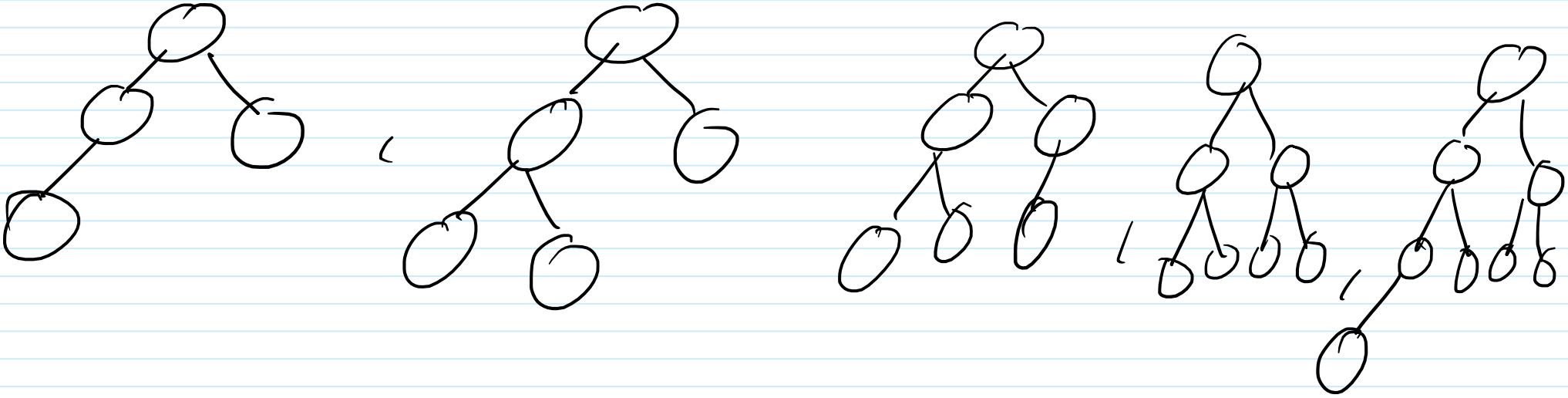
Definicja (kopiec jako struktura logiczna).

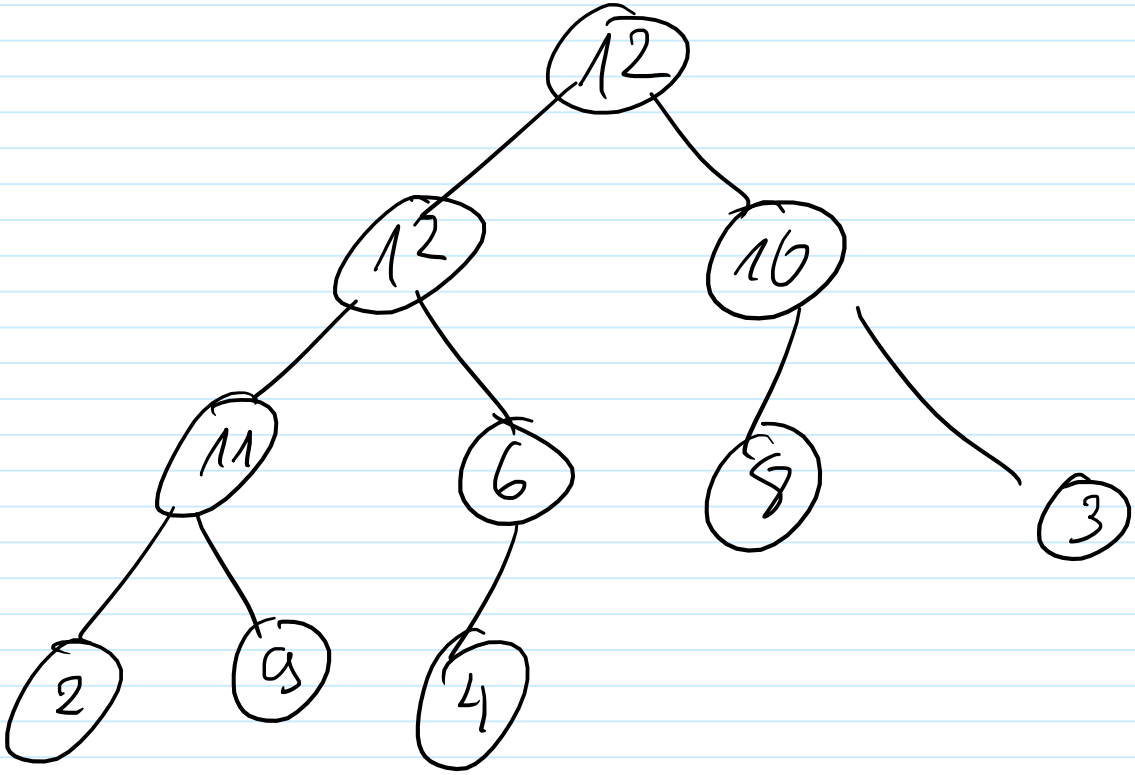
Kopiec to drzewo binarne t. ż.

1. rośnie poziomami czyli kopiec jest pełnym drzewem binarnym, z wyjątkiem (ewentualnie) ostatniego poziomu, który rośnie od lewej do prawej

② Dla każdego elementu kopca v, klucz v jest większy lub równy od klucza jego dzieci (jeśli istnieją).



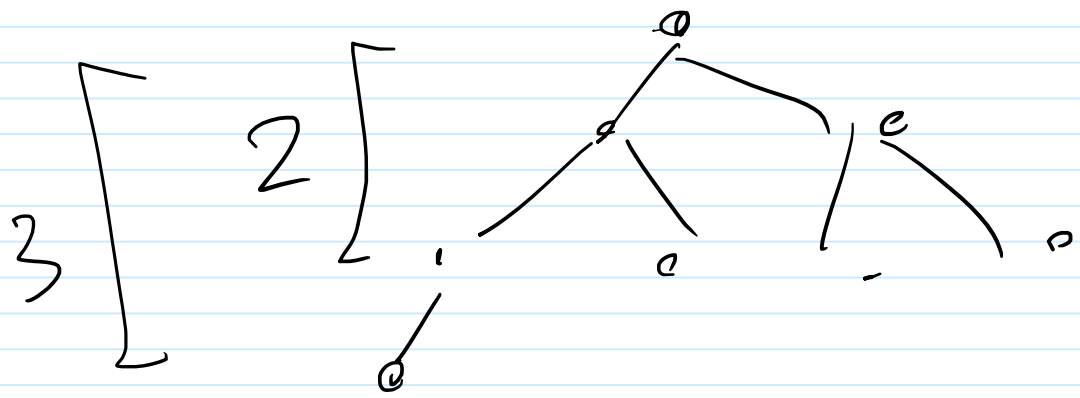




Wielkość kopca to liczba jego elementów.

Fakt

Jeśli kopiec ma n elementów, to jego głębokość jest $\leq \lfloor \log_2(n) \rfloor$, $n \geq 0$.



Jak zaimplementować kopiec?

Kluczowe dla nas będą dwie operacje.

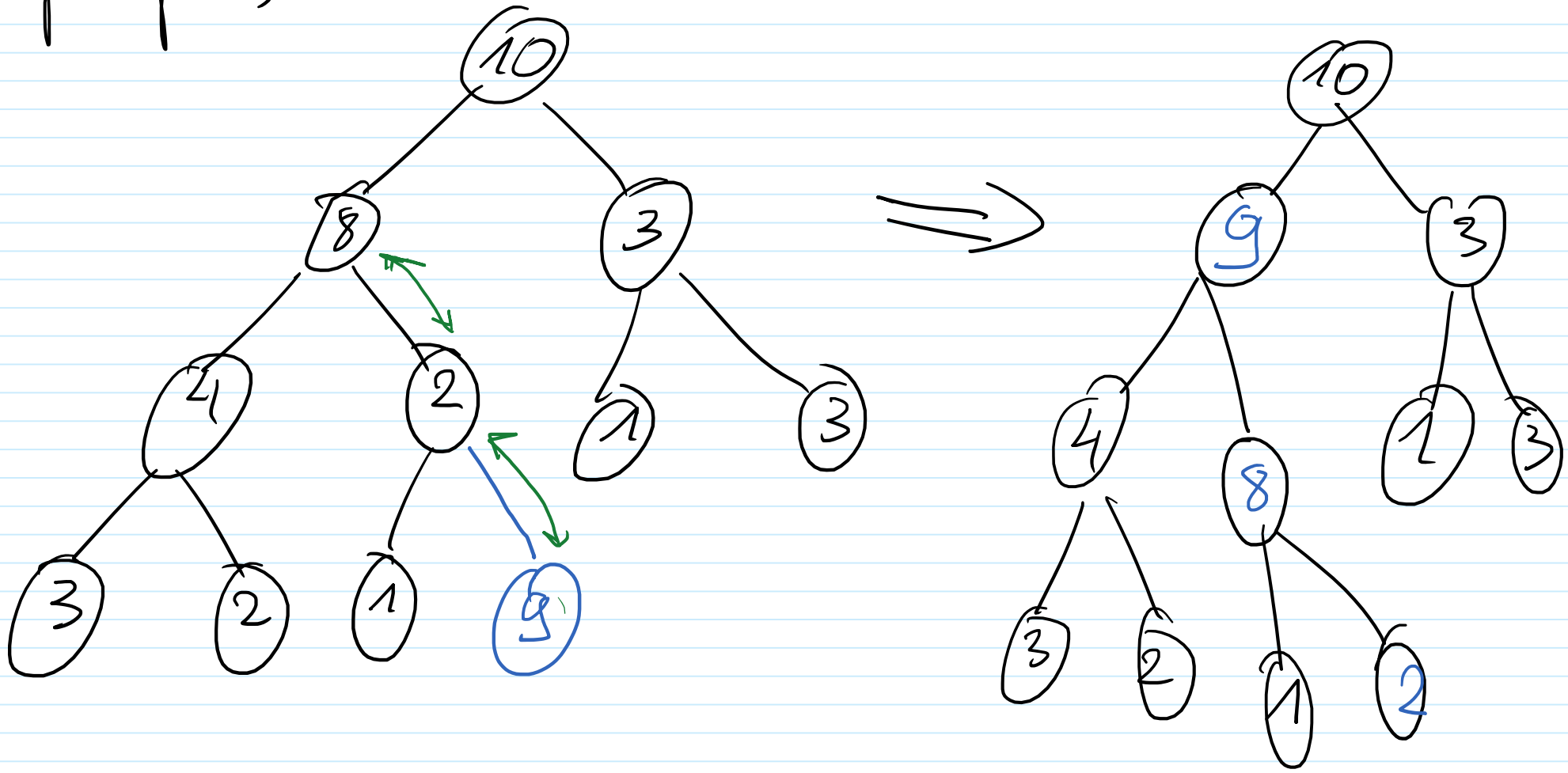
upheap, downheap.

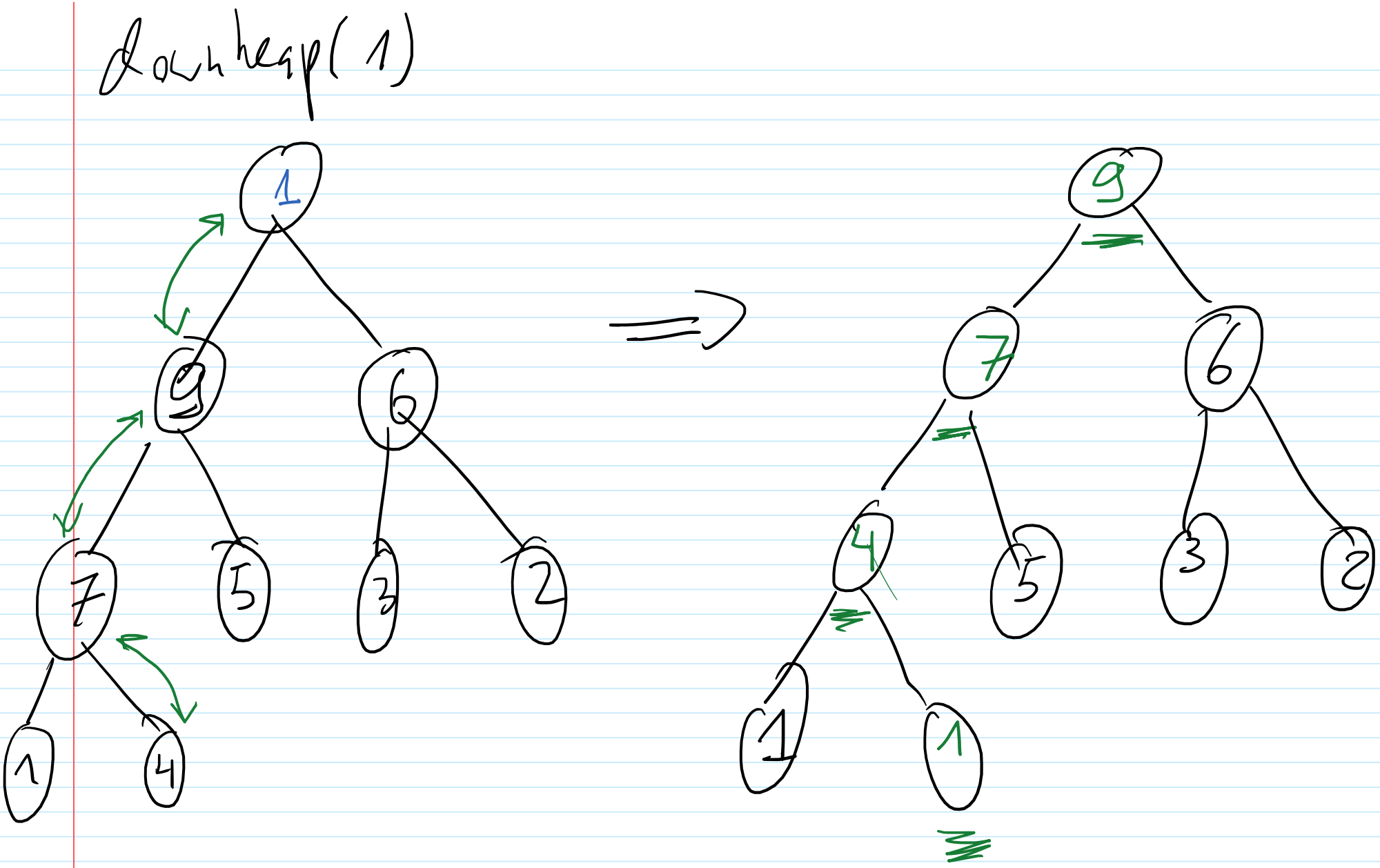
upheap(e) – wyobrażamy sobie, że priorytet elementu e się zwiększył.

upheap(e) przenosi element e wyżej

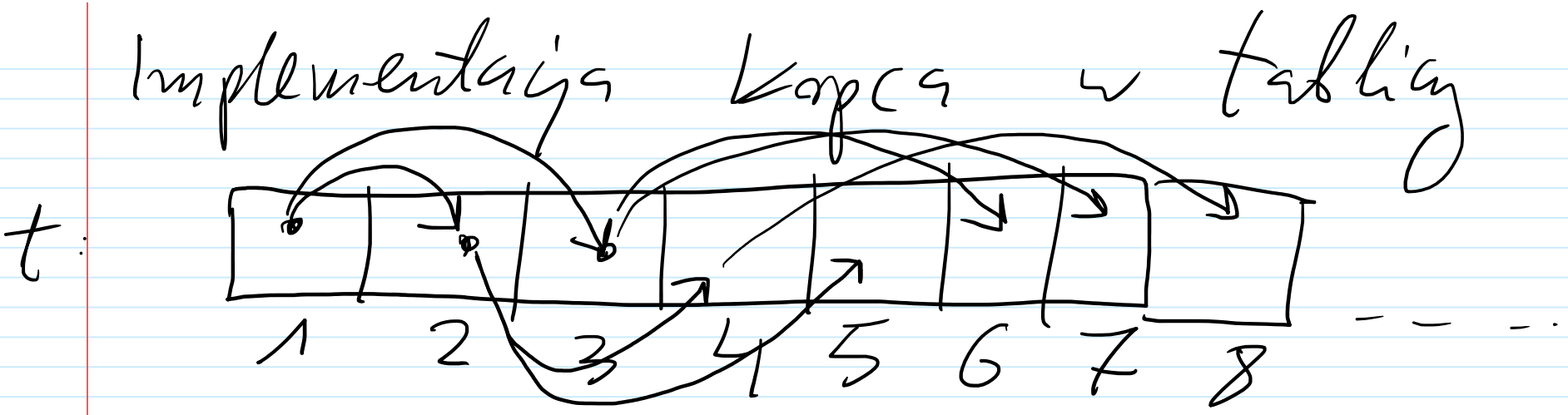
downheap(e) – jeśli priorytet e się zmniejszył, to downheap(e) spycha e w dół kopca.

upheap(9)




downheap(1)
1
9
6
7
5
3
2
1
4
9
7
6
4
5
3
2
1
1

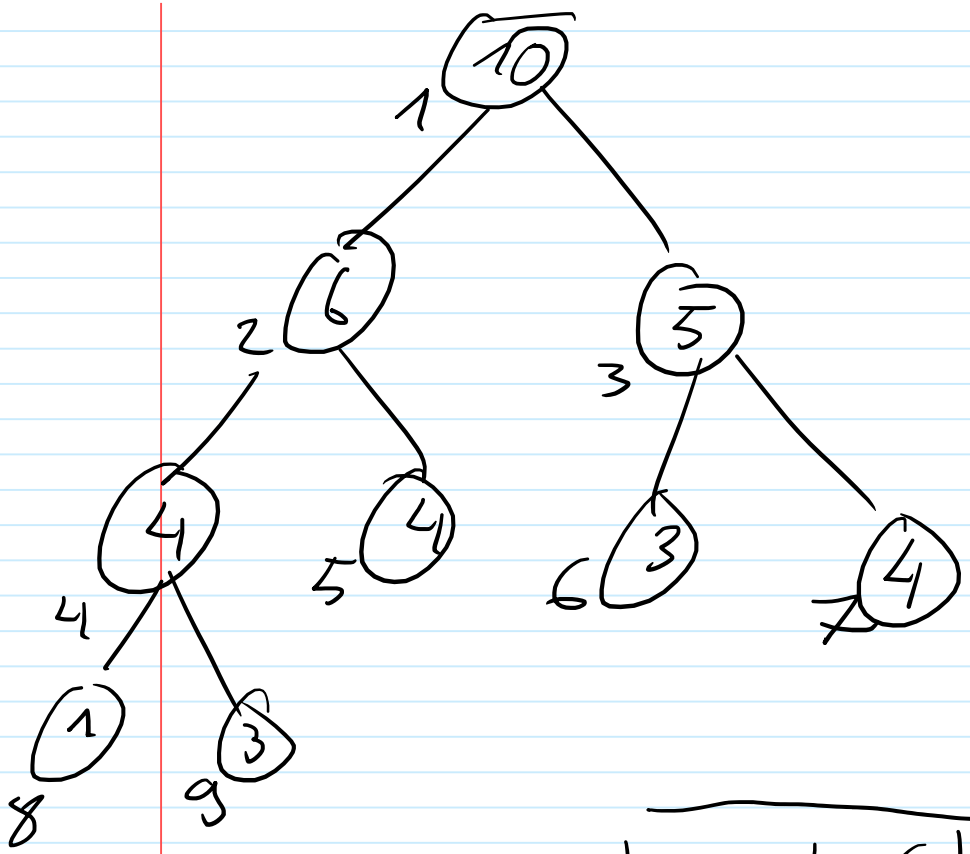
Implementacja kopca w tablicy

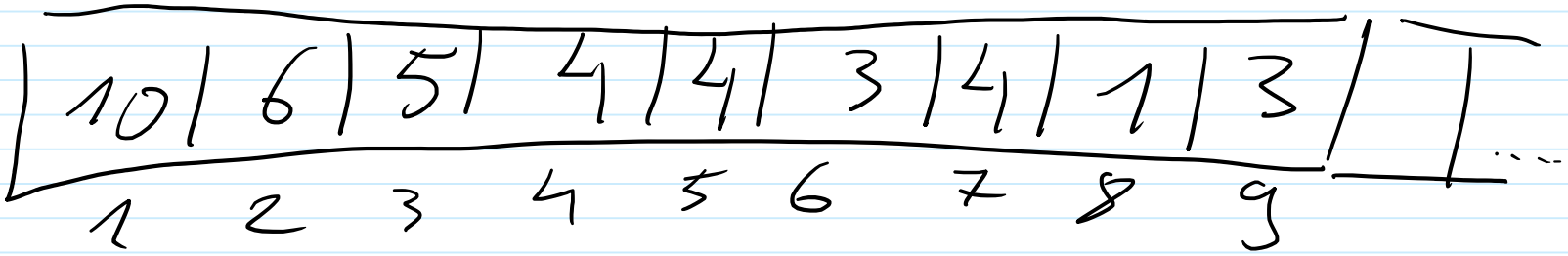


W implementacji kopca t, t[1] to korzeń.

Dla elementu i dzieci t[i] to t[2i], t[2i+1],

rodzic to $t[\lfloor \frac{i}{2} \rfloor]$.


10
6
5
4
4
3
4
1
3
1
2
3
4
5
6
7
8
9


10 6 5 4 4 3 4 1 3
1 2 3 4 5 6 7 8 9

Kopiec implementujemy w tablicy

int h[1..N];

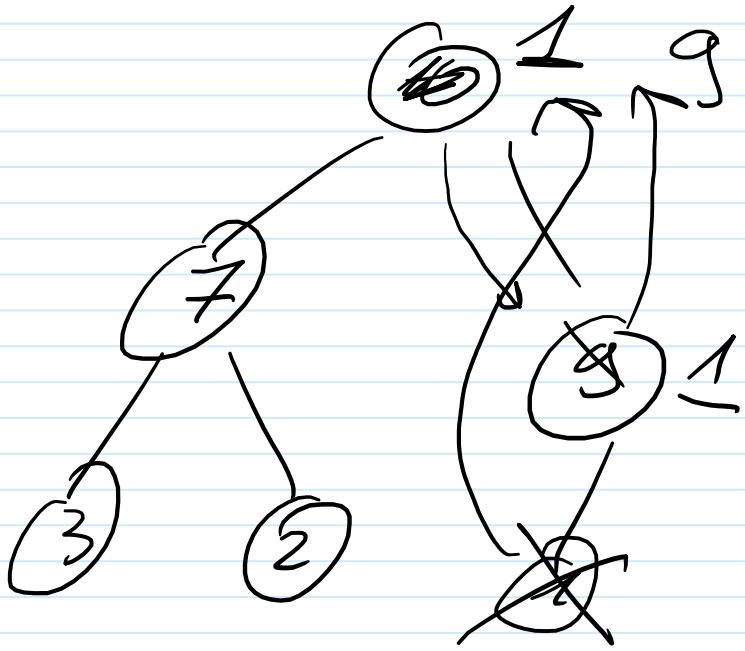
int size = 0; - ilość elementów w kopcu.

Zaimplementujemy operacje na kopcu na tablicy.

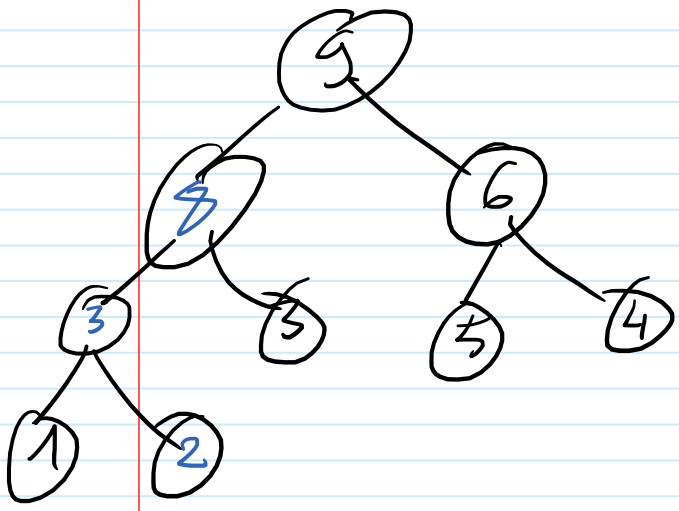
```
upheap(int h[], int i){
  // przesuwamy element h[i]
  while (i > 1){
    if (h[⌊i/2⌋] < h[i]){
      swap(h, ⌊i/2⌋, i);
      i := ⌊i/2⌋;
    }
    else i := 0;
}
```

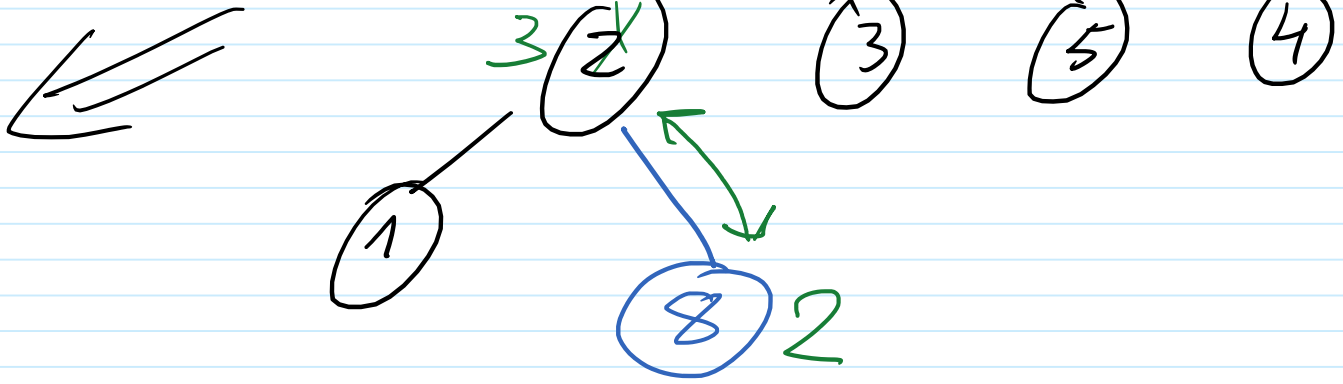
```
downheap(int h[], int i, int size) {
// sprychamy element t[i] w dół kopca
while( i <= ⌊size/2⌋ ) { // 2i <= size
                          h[i] ma dzieci
  if(2i >= size) then k = 2·i;
  else { // jest 2 dzieci
    if(h[2i] > h[2i+1]) then k = 2·i;
    else k = 2·i+1;
  }
  if( h[i] < h[k] ) then {swap(h, i, k); i := k;}
  else i = size+1;
} }
```

```
int delTop(){
    int pom = h[1];
    h[1] = h[size];
    size = size-1;
    downheap(h, 1, size);
    return pom;
}
```



```
insert(int h[], int i) {
    size := size + 1;
    h[size] := i;
    upheap(h, i);
}
```





Kontynuacja wykładu, cel: sortowanie przez kopcowanie (heapsort). 2022.04.07

Przypomnienie:

Implementujemy kolejkę priorytetową.
Podstawowe operacje:

- enqueue(x,priorytet) - zakolejkowanie elementu o priorytecie priorytet,
- Elem delMax() - zdjęcie z kolejki elementu o największym priorytecie,
- top() - sprawdzenie jaki jest pierwszy element w kolejce,
- empty() - sprawdzenie pustości kolejki

Operacje te zaimplementowaliśmy w tablicy t[1..N] z pomocniczą zmienną heapsize, która określa rozmiar kopca.

Elementy t[1..heapsize] to elementy kopca.
Kopiec w tablicy traktujemy jako drzewo binarne.
Dziećmi elementu t[i] są elementy t[2*i], t[2*i+1], o ile 2*i, 2*i<=heapsize.
Rodzic elementu t[i] to element t[podloga(i/2)], o ile i>1.

Musimy napisać operację poprawiania kopca w górę i w dół.

```
//N - rozmiar sterty gdzie t[N] to ostatni
element
//    t[1] - element na wierzcholku sterty

void downheap(int *t,int i, int N){
      int j=2*i;
      while(j <=N){
            if((t[j]<t[j+1])&& j+1<=N)
                  j++;
            if(t[i]<t[j]){
                  swap(t,i,j);
                  i=j;
                  j=2*i;
            }
            else
                  j=N+1;
      }
}
```

```
void swap(int *t, int i, int j){
      int pom=t[i];
    t[i]=t[j];
      t[j]=pom;
}
```

```
  void downheap_rec(int *t,int i, int N){
        int j=2*i;
        if(j>N) return;
        if(j+1<=N)
            if(t[j]<t[j+1]) j=j+1;
        if(t[i]<t[j]){
              swap(t,i,j);
              downheap_rec(t,j,N);
        }
  }
```

Poprawianie kopca - gety

```
void upheap(int *t, int i){
    while(i>1)
        if(t[i]>t[i/2]){
            swap(t,i,i/2);
            i=i/2;
            }
        else
            i=1;
}
```

```
void upheap(int *t, int i){
    if(i<=1) return;
    if(t[i]>t[i/2]) {
        swap(t,i, i/2);
        upheap(t, i/2);
    }
}
```

delMax()

Mamy globalną tablicę int heap[N]; int heapsize=0;

```
int delMax(){
      int tmp=heap[1];
      heap[1]=heap[heapsize];
      heapsize=heapsize-1;
      downheap(1,heapsize);
     return tmp;
}
```

enqueue(int)

```
void enqueue(int k){
     if(heapsize+1>=N) return;
    heapsize=heapsize+1;
    t[heapsize]=k;
    upheap(t,k);
}
```

HeapSort

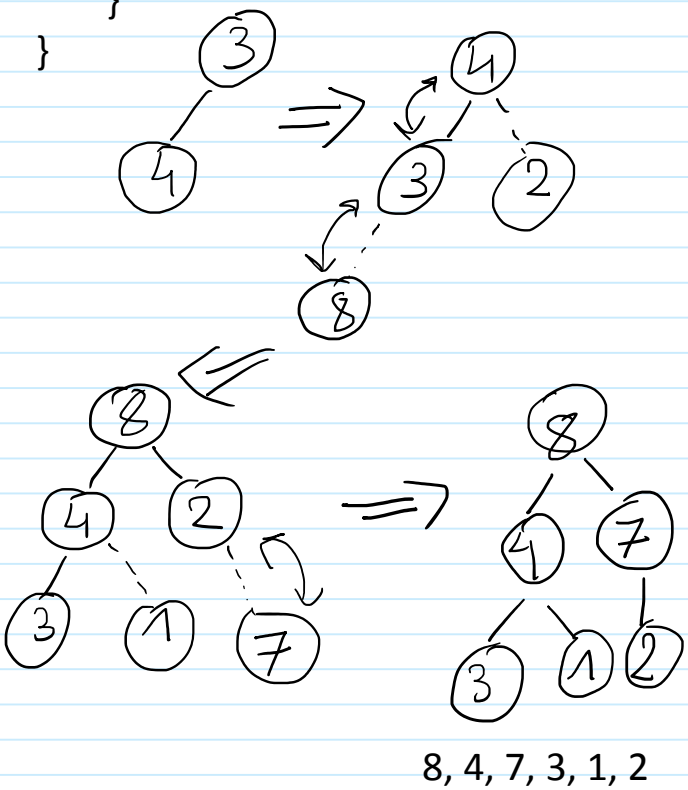
```
void heapsort1(int *t, int N){
    for(int k=2;k<=N;k++)
        upheap(t,k);
    while(N>1){
        swap(t,1,N);
        N--;
        downheap(t,1,N);
    }
}
```

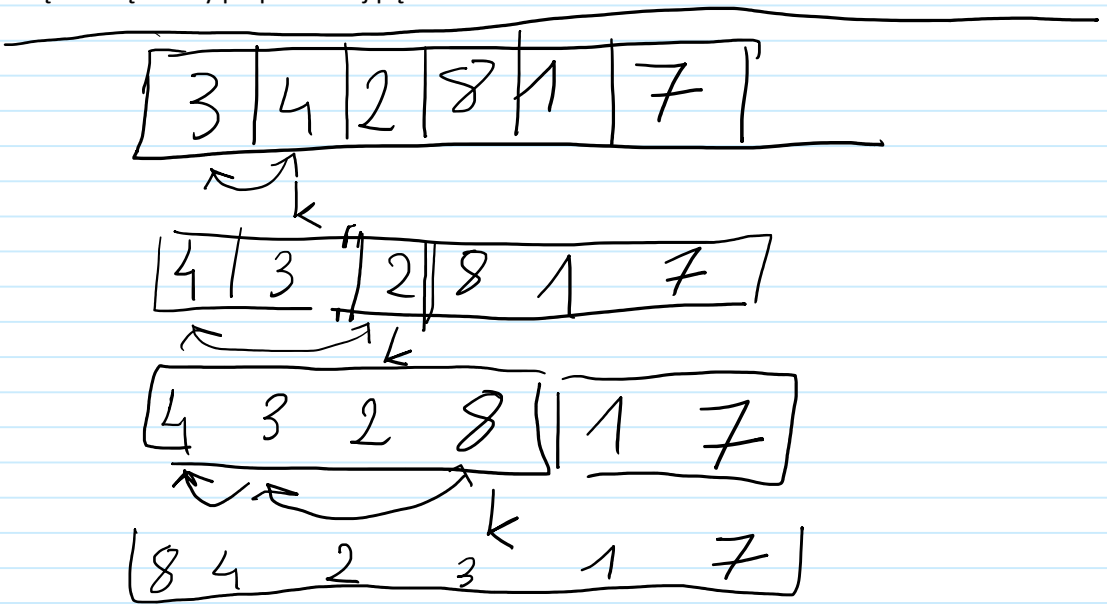
Sortujemy tablicę t[1,…,N]
Etap 1. Tworzymy kopiec.
Etap 2. Usuwamy kopiec wstawiając po kolei
elementy na właściwe miejsca
na koniec tablicy.

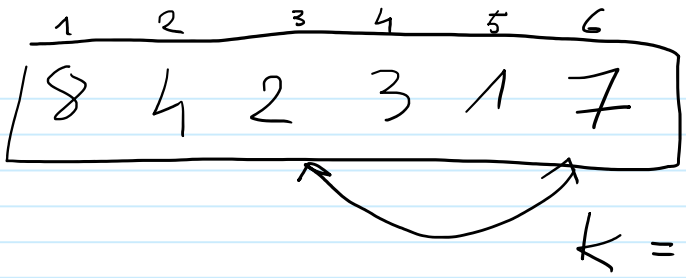
t[1..6]: 3, 4, 2, 8, 1, 7

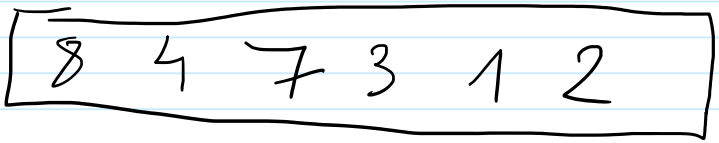
1. Tworzymy kopiec
   4, 3, 2, 8, 1, 7
   dokładamy 2 i jest ok
   dokładamy 8 (k=4) i poprawiamy kopiec w górę
   8, 4, 2, 3, 1, 7
   kopiec to 8, 4, 2, 3, a 1, 7 jest poza kopcem
   dokładamy 1 - ok.
   dokładamy 7 i poprawiamy kopiec w górę otrzymując
   8, 4, 7, 3, 1, 2

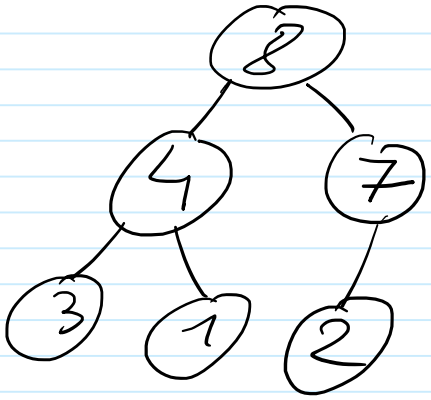
Taką tablicę mamy po pierwszej pętli.





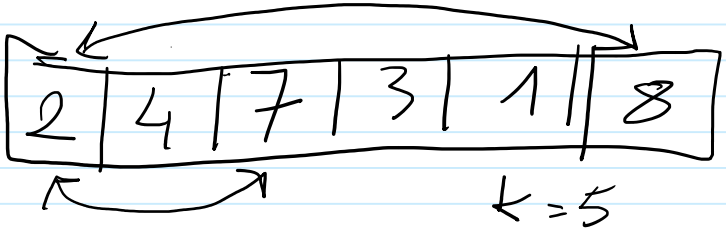




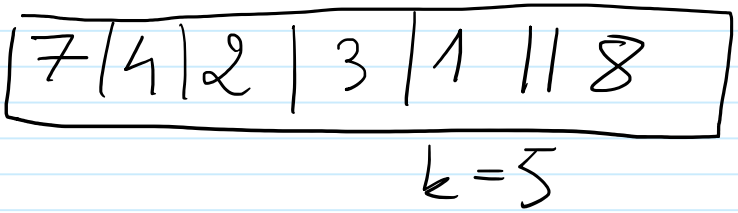


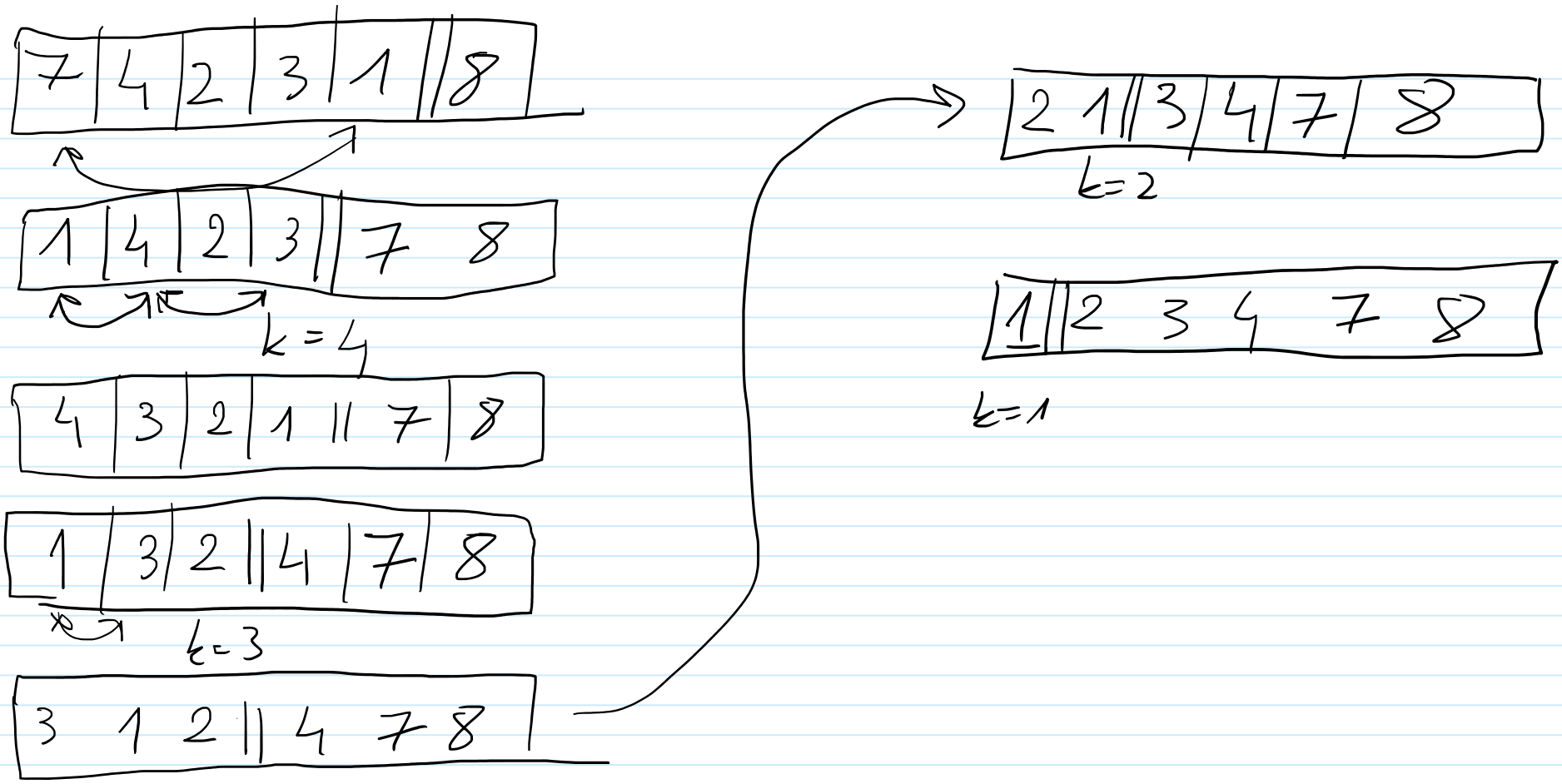
Teraz, usuwamy po jednym elemencie z kopca wstawiając go na jego miejsce w tablicy.
Usuwamy element z wierzchołka kopca ale jest to element maksymalny, więc trzeba wstawić go na koniec tablicy.

k - rozmiar kopca, k = 6



Poprawa kopca w dół, bo wierzchołek się zmniejszył.




7 4 2 3 1 8
1 4 2 3 7 8
k = 4
4 3 2 1 7 8
1 3 2 4 7 8
k = 3
3 1 2 4 7 8
2 1 3 4 7 8
k = 2
1 2 3 4 7 8
k = 1

Heapsort - własności

① Działa w pesymistycznym czasie $O(n \log n)$

② Używa stałej pamięci (bo nie ma rekursji i struktur pomocniczych)

③ Wykonuje jednak sporo przestawień i porównań (stała ukryta w notacji $O(.)$ nie jest "bardzo mała")

Ad 1

Czas pesymistyczny $O(n \log_2(n))$.

```
void heapsort1(int *t, int N){
        for(int k=2;k<=N;k++)
                upheap(t,k);
    while(N>1){
                swap(t,1,N);
                N--;
                downheap(t,1,N);
        }
}
```

Wykonujemy tę pętlę N-razy koszt jednego wykonania to koszt upheap który po prostu wsadza ścieżkę w drzewie $\leq O(\log_2 N)$.

W sumie $O(N \cdot \log_2 N)$

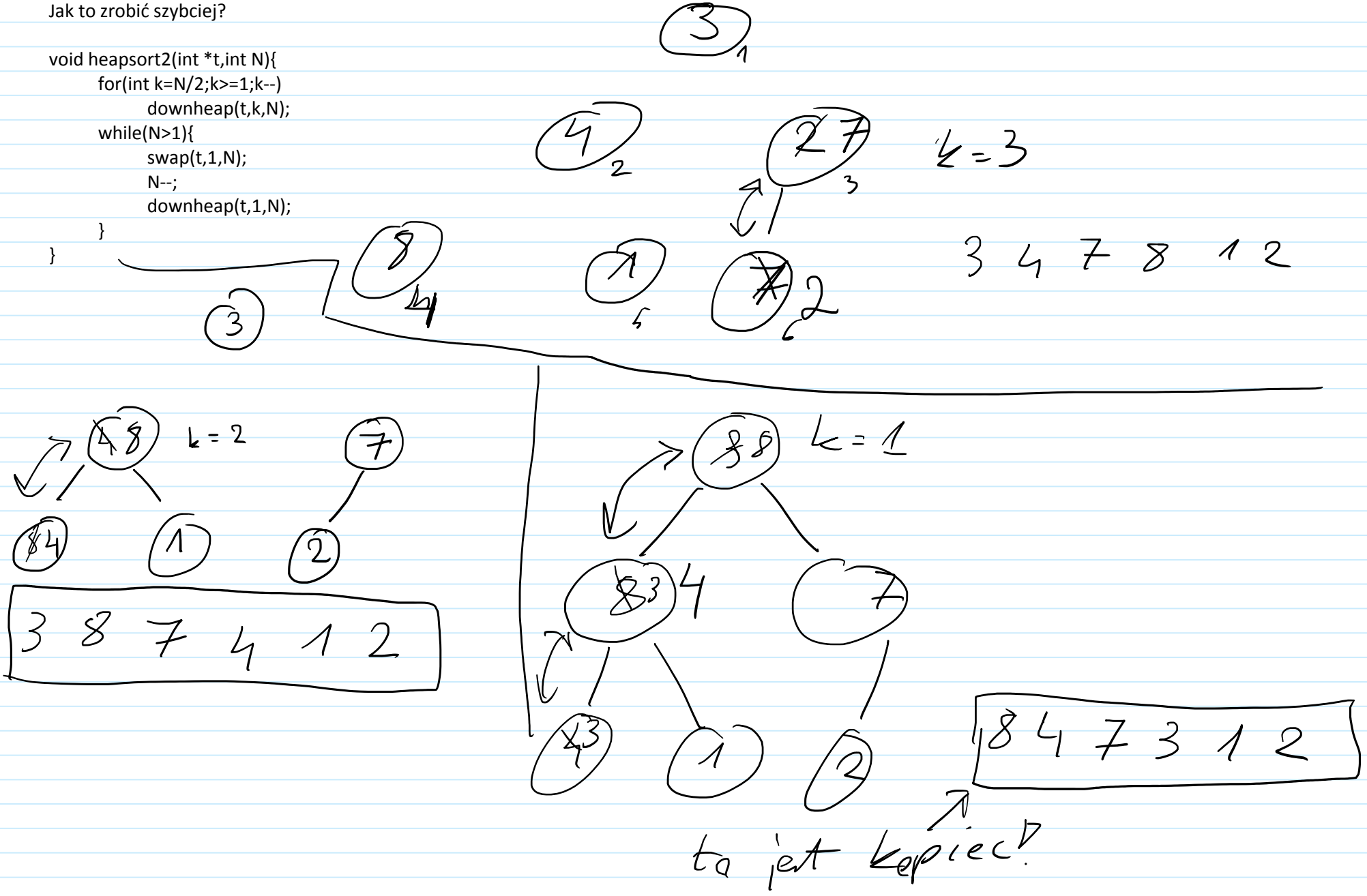
Wykonujemy N razy koszt "down heap" to $\leq O(\log_2 N)$.

W sumie koszt "while" to $O(N \log_2 N)$.

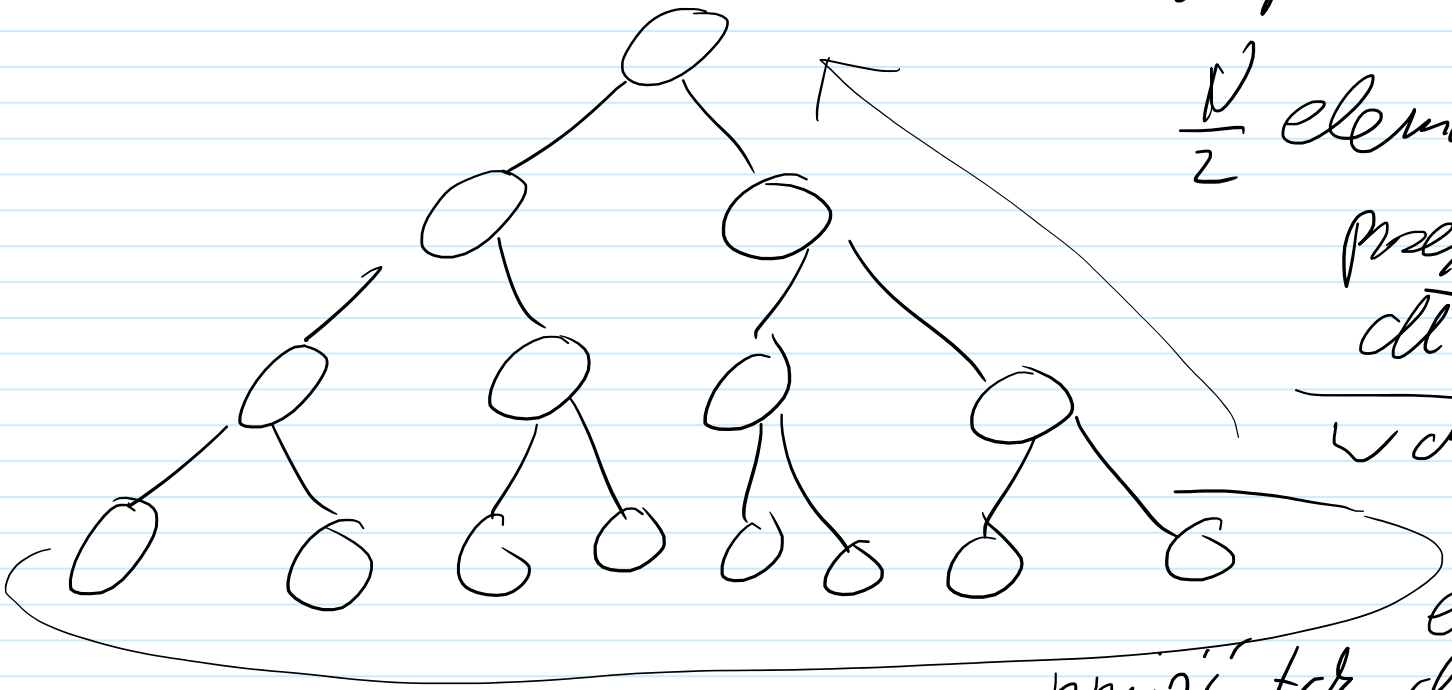
W sumie mamy $O(N \log_2 N)$ – koszt dwóch pętli.

Jak to zrobić szybciej?

```
void heapsort2(int *t,int N){
        for(int k=N/2;k>=1;k--)
                downheap(t,k,N);
        while(N>1){
                swap(t,1,N);
                N--;
                downheap(t,1,N);
        }
}
```

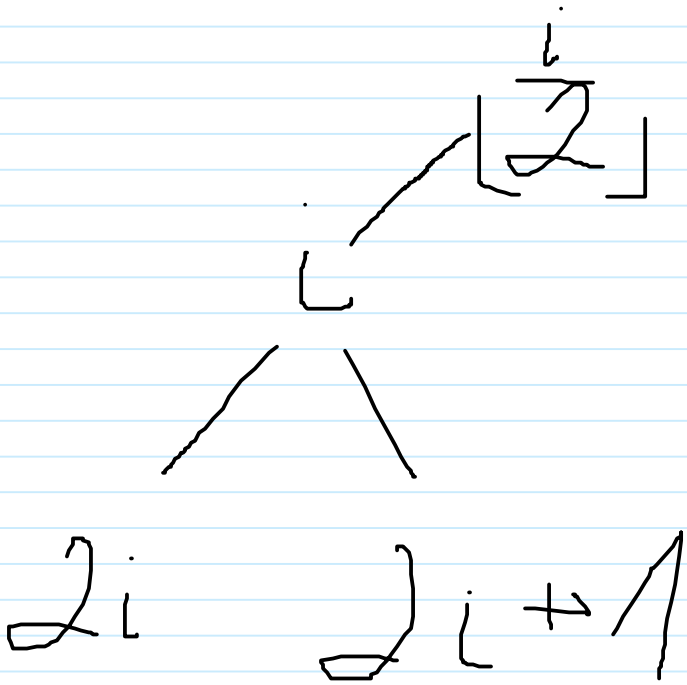


Co w ten sposób zyskujemy?
Czas tworzenia takiego kopca jest liniowy.



W pierwszej wersji $\frac{N}{2}$ elementów może przejść drogę dł. $\log_2 N$

W drugiej wersji tylko jeden element może przejść tak długą drogę.


i
⌊i/2⌋
2i
2i+1