Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Wykład 8.

Kolejki priorytetowe

Kolejki priorytetowe

Kolejka priorytetowa - to najbardziej ogólny rodzaj kolejki.

Elementy takiej kolejki pobierane są w kolejności:

1. Wyznaczonej wartością priorytetu elementu w kolejce (jest to dodatkowa cecha elementów wprowadzająca kolejność ich pobierania z kolejki); zakładamy, że pobieramy element o najwyższym priorytecie.

2. Przy równych priorytetach pobierany jest element wcześniej umieszczony w kolejce.

Zwykła kolejka jest szczególnym przypadkiem kolejki priorytetowej, w której wszystkie elementy mają priorytet określony przez czas ich pobytu w kolejce.

Implementacja kolejki jest uzależniona od charakteru zbioru wartości priorytetów.

Rodzaje kolejek priorytetowych:

Kolejki z nieograniczonym zbiorem wartości priorytetów:

- nieuporządkowane,
- uporządkowane,
- o organizacji stogowej.

Kolejki z ograniczonym (niewielkim) zbiorem wartości priorytetów.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

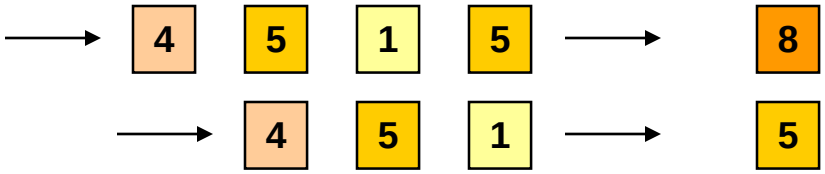
Kolejki priorytetowe z nieograniczonym zbiorem wartości priorytetów

Kolejki nieuporządkowane

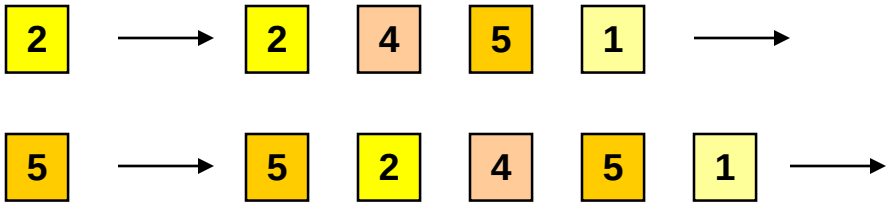
W takich kolejkach:
- wstawianie jest dokonywane na koniec kolejki (złożoność O(1)),
- pobranie i usunięcie elementu o najwyższym priorytecie (pierwszego z równych); wymaga wyszukania tego elementu (złożoność O(N)).

Przykład:
- stan kolejki w danej chwili:

- stan kolejki po kolejnych pobraniach i usunięciach elementu:



- stan kolejki po kolejnych dopisaniach:



Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Przykładowa implementacja kolejki priorytetowej oparta na liście nieuporządkowanej (na podstawie: [Harris, Ross])

```
package queues;
import lists.LinkedList;
import lists.List;
import sorting.Comparator;

public class UnsortedListPriorityQueue implements Queue {
   private final List _list;
   private final Comparator _comparator;  // do "obsługi" priorytetu

   public UnsortedListPriorityQueue(Comparator comparator) { // konstruktor
       _comparator = comparator;
       _list = new LinkedList();
   }

   public void enqueue(Object value) {
      _list.add(value);
   }

   public Object dequeue() throws EmptyQueueException {
      if (isEmpty()) throw new EmptyQueueException();
      return _list.delete(getIndexOfLargestElement());
   } // c.d.n.
```

Przykładowa implementacja kolejki nieuporządkowanej oparta na liście (na podstawie: [Harris, Ross])

```
// c.d.:

  private int getIndexOfLargestElement() { // pobranie indeksu elementu o najwyższym
                                           // priorytecie
      int result = 0;
      for (int i = 1; i < _list.size(); ++i)
        if (_comparator.compare(_list.get(i), _list.get(result)) > 0)
          result = i;
      return result;
  }

  public void clear() {
      _list.clear();
  }

  public int size() {
      return _list.size();
  }

  public boolean isEmpty() {
      return _list.isEmpty();
  }
}
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

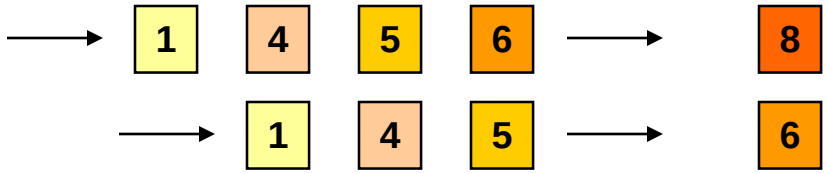
Kolejki priorytetowe z nieograniczonym zbiorem wartości priorytetów

Kolejki uporządkowane

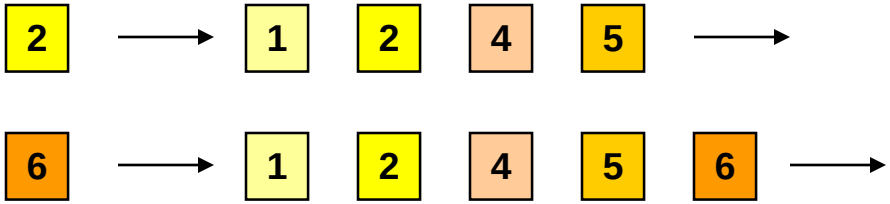
Wykorzystują uporządkowaną tablicę lub listę; wówczas:
- wstawianie jest dokonywane na właściwe miejsce (złożoność O(N)),
- pobranie i usunięcie elementu (o najwyższym priorytecie) z czoła kolejki (złożoność O(1)).

Przykład:
- stan kolejki w danej chwili:

- stan kolejki po kolejnych pobraniach i usunięciach elementu:



- stan kolejki po kolejnych dopisaniach:



Przykładowa implementacja kolejki priorytetowej oparta na liście uporządkowanej (na podstawie: [Harris, Ross])

```
package queues;
import lists.LinkedList;
import lists.List;
import sorting.Comparator;

public class SortedListPriorityQueue implements Queue {
   private final List _list;
   private final Comparator _comparator;

   public SortedListPriorityQueue(Comparator comparator) {
       _comparator = comparator;
       _list = new LinkedList();
   }

   public void enqueue(Object value) {
      int pos = _list.size();
      while (pos > 0 && _comparator.compare(_list.get(pos - 1), value) > 0)
           --pos;
       _list.insert(pos, value);
   }
} // c.d.n.
```

Przykładowa implementacja kolejki priorytetowej oparta na liście uporządkowanej (na podstawie: [Harris, Ross])

```
// c.d.:

  public Object dequeue() throws EmptyQueueException {
      if (isEmpty()) throw new EmptyQueueException();
      return _list.delete(_list.size() - 1);
  }

  public void clear() {
      _list.clear();
  }

  public int size() {
     return _list.size();
  }

  public boolean isEmpty() {
     return _list.isEmpty();
   }
}
```

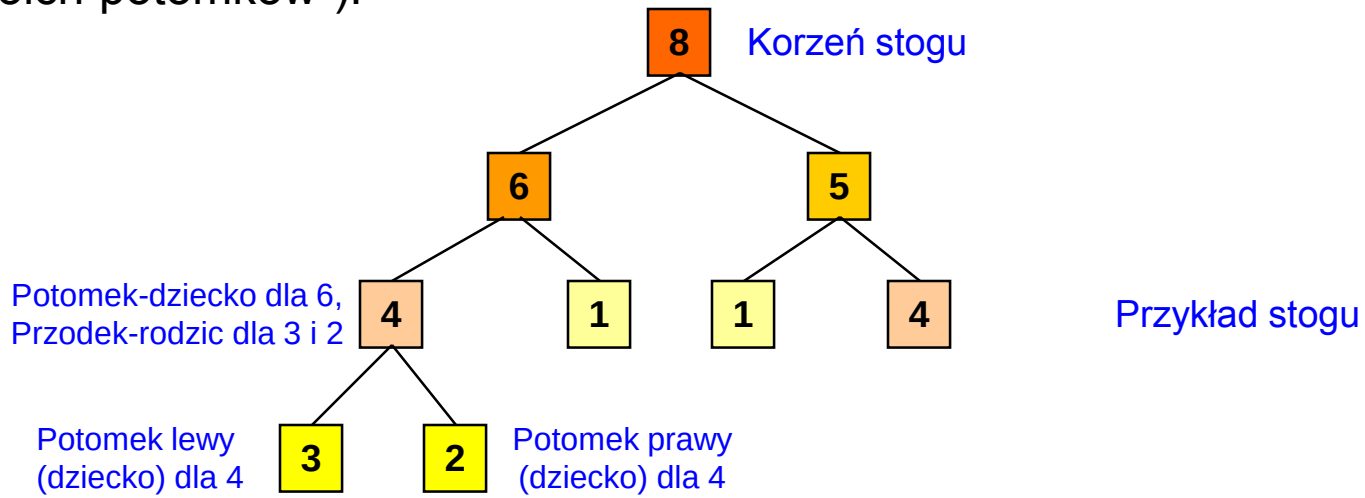
Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Kolejki priorytetowe z nieograniczonym zbiorem wartości priorytetów

Kolejki o organizacji stogowej

Wykorzystują stóg (*ang. Heap*), czasem nazywany też kopcem lub stertą.
Koncepcja stogu:
Stóg jest drzewem binarnym, posiadającym własności:

- kształtu: jest pełnym drzewem, przy czym ostatni poziom jest zapełniany od lewej strony ku prawej,
- uporządkowania (tzw. warunek stogowy): wartość węzła jest zawsze większa od wartości każdego z jego potomków (w skrócie: "węzeł jest zawsze większy od swoich potomków").

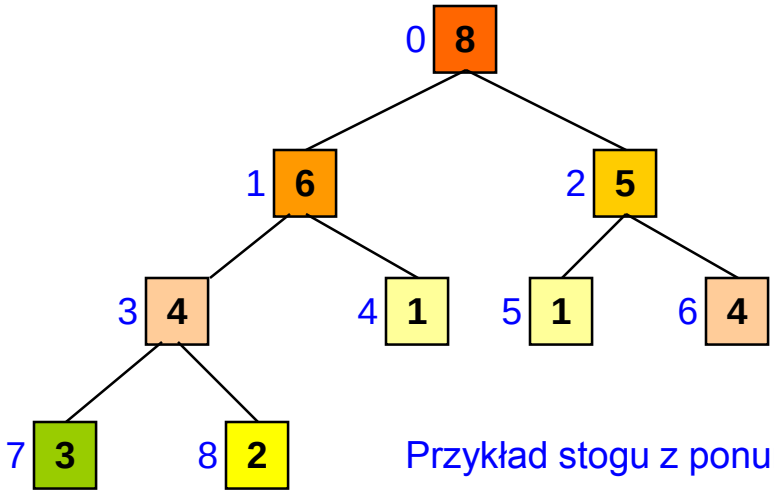


Własność uporządkowania jest przechodnia (w sensie matematycznym), więc korzeń stogu ma wartość największą ("jest największym elementem stogu").
Uwaga: Stogu nie należy utożsamiać z listą posortowaną!

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

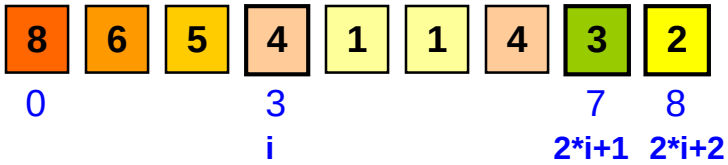
Kolejki priorytetowe o organizacji stogowej

Ze stogu można korzystać tak, jak z listy implementującej interfejs List.
Można ponumerować wierzchołki stogu według zasady:
- korzeń otrzymuje numer 0,
- na kolejnych poziomach (z góry na dół) kolejne wierzchołki (od lewej do prawej) otrzymują kolejne numery: 1, 2, 3, ...



Przykład stogu z ponumerowanymi węzłami

Wówczas lista odwzorowująca stóg ma postać:



Lewy potomek (dziecko) elementu z i-tej pozycji znajduje się na pozycji 2*i+1, prawy – na pozycji 2*i+2, przodek-rodzic elementu i-tego jest na pozycji (i-1)/2 (z zaokrągleniem w dół). Element na pozycji 0 nie posiada przodka-rodzica.

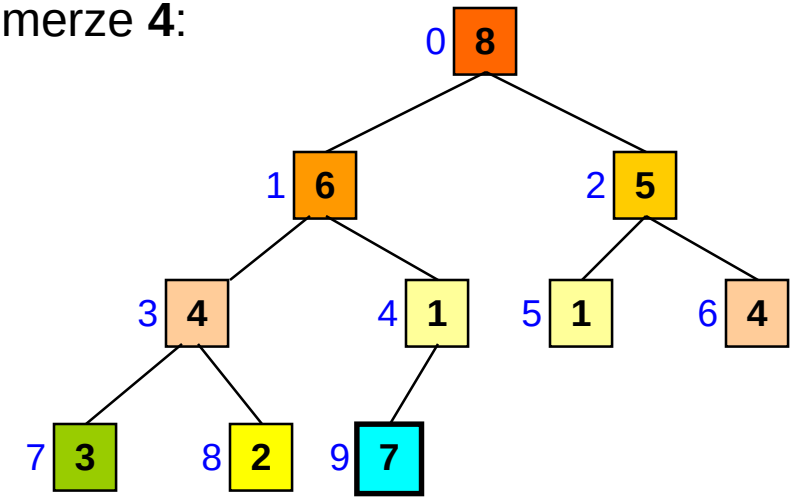
Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Operacje na stogu

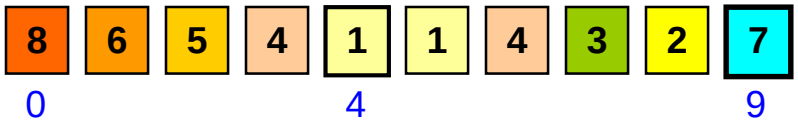
Operacje na stogu (dodanie elementu do stogu oraz pobieranie i usuwanie elementu ze stogu) muszą zapewnić zachowanie warunku stogowego.
Operacja dodania elementu do stogu obejmuje dwa kroki:

1. **Dołączamy element do pierwszego (w sensie numeracji) węzła, który nie ma "kompletu" potomków-dzieci.**

Przykład: dodawany węzeł o wartości **7** staje się lewym potomkiem węzła o numerze **4**:



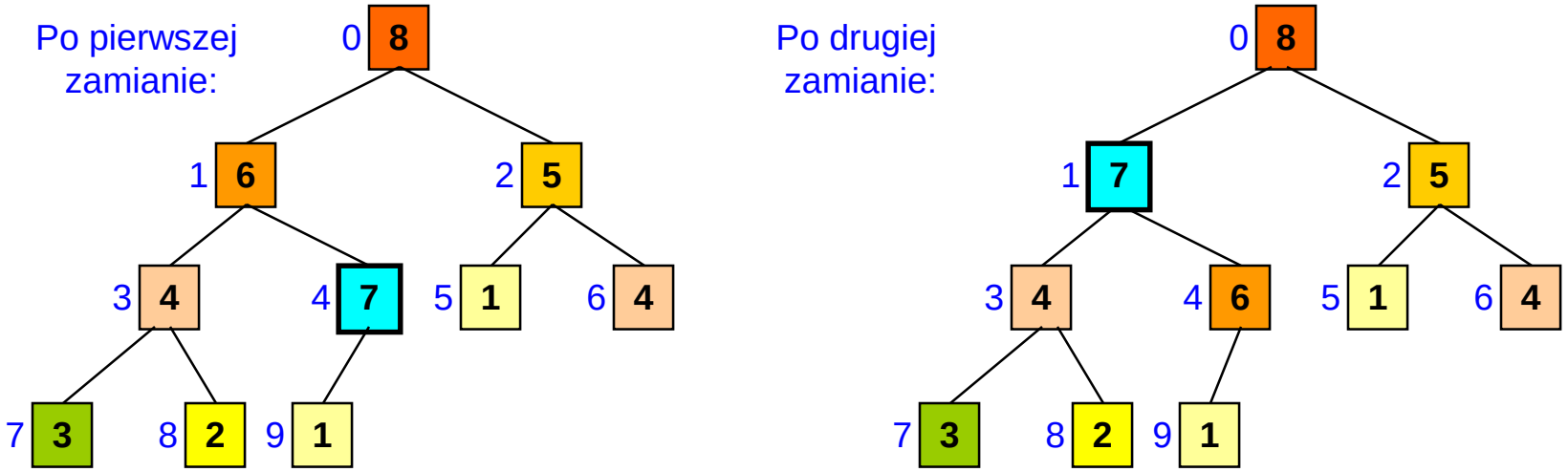
a w liście zajmuje ostatnią pozycję (zostaje dołączony na końcu listy):



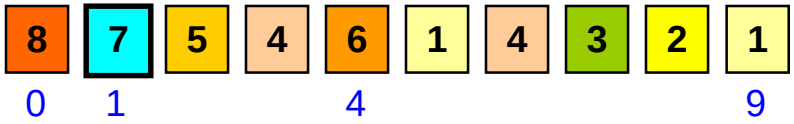
Dodanie tego węzła prowadzi do naruszenia warunku stogowego: węzeł nr 4 (o wartości 1) ma potomka o większej wartości (równej 7).
Przywrócenie warunku stogowego wymaga przeniesienia dopisanego elementu na wyższy poziom, tzw. wynurzenia (wyniesienia) elementu).

Operacje na stogu: dodanie elementu do stogu

2. Realizujemy wynurzenie (wynoszenie) elementu; polega to na sukcesywnej zamianie miejscami potomka-dziecka i jego przodka-rodzica (z dołu do góry stogu), aż do uzyskania sytuacji, w której zostanie spełniony (przywrócony) warunek stogowy.



Zamiany miejscami wierzchołków podczas wynurzania elementu dołączanego do stogu – to zamiany miejscami elementów w liście. Postać listy po wynurzeniu elementu:

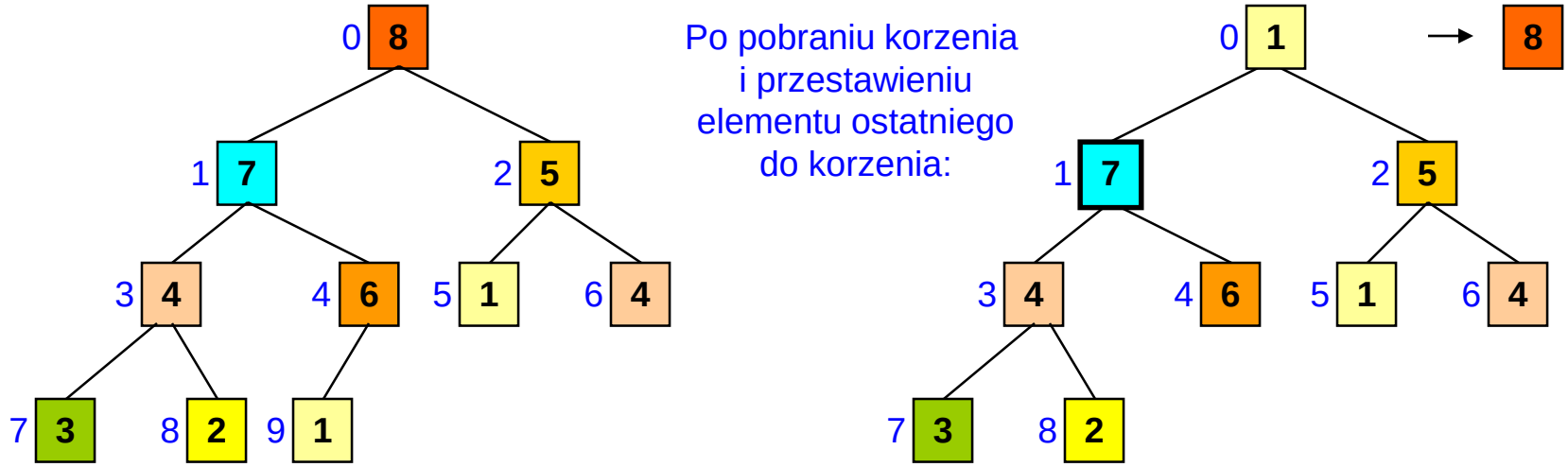


Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

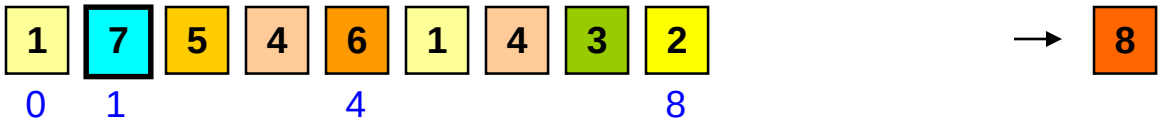
Operacje na stogu: pobieranie i usuwanie elementu ze stogu

Operacja pobrania i usunięcia elementu ze stogu obejmuje dwa kroki:

1. **Pobieramy element z korzenia stogu (jako największy) i przenosimy ostatni element stogu (w jego listowym rozwinięciu) w miejsce pobranego korzenia.**



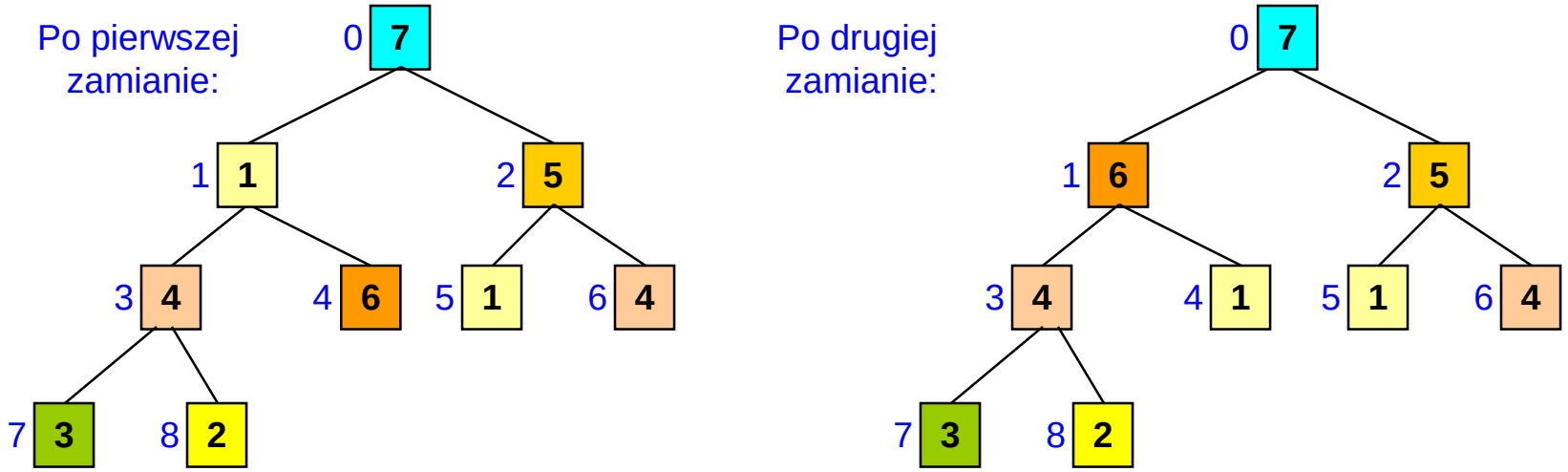
Rozwinięcie listowe stogu ma teraz postać:



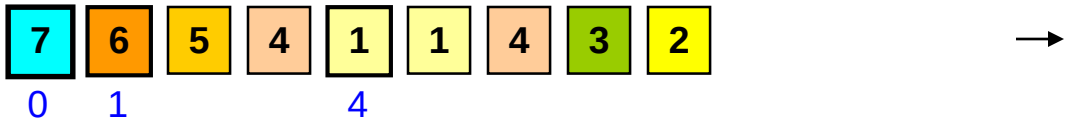
Przestawienie naruszyło warunek stogowy. Element z wierzchołka musi zostać przeniesiony na niższy poziom stogu (zatopiony, opuszczony), by przywrócić warunek stogowy.

Operacje na stogu: pobieranie i usuwanie elementu ze stogu

2. Realizujemy zatapianie (opuszczanie) elementu; jest to sukcesywna zamiana tego elementu z większym z jego potomków-dzieci – aż do spełnienia warunku stogowego.



Zatapianie elementu – to zamiany miejscami elementów w liście (tu: 0<->1, 1<->4).
Postać listy po zatopieniu (opuszczeniu) elementu:



Przedstawione operacje na stogu wykorzystuje implementacja stogowej kolejki priorytetowej.

Przykładowa implementacja stogowej kolejki priorytetowej

```
package queues;
import lists.ArrayList;
import lists.List;
import sorting.Comparator;

public class HeapOrderedListPriorityQueue implements Queue {
    private final List _list;
    private final Comparator _comparator;

    public HeapOrderedListPriorityQueue(Comparator comparator) {
        _comparator = comparator;
        _list = new ArrayList();
    }

    public void enqueue(Object value) { // dołączenie elementu
        _list.add(value);
        swim(_list.size() - 1);                 // wynurzenie (wyniesienie) elementu
    }

// c.d.n.
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Przykładowa implementacja stogowej kolejki priorytetowej

```
// c.d.:
 private void swim(int index) { // wynurzanie elementu (wynoszenie elementu w górę)
    int parent;
    while(index != 0 &&
             _comparator.compare(_list.get(index), _list.get(parent= (index - 1) / 2)) > 0)
        { swap(index, parent);
          index=parent; }   //  wersja iteracyjna; w [Harris, Ross] jest wersja rekurencyjna
 }
 private void swap(int index1, int index2) { // zamiana miejscami dwóch elementów
     Object temp = _list.get(index1);
     _list.set(index1, _list.get(index2));
     _list.set(index2, temp);
 }
public Object dequeue() throws EmptyQueueException { // pobranie/usunięcie elementu
    if (isEmpty()) throw new EmptyQueueException();
    Object result = _list.get(0);
    if (_list.size() > 1) {
        _list.set(0, _list.get(_list.size() - 1));
        sink(0);                                          // zatapianie (opuszczanie) elementu
    }
    _list.delete(_list.size() - 1);
    return result;
} // c.d.n.
```

Przykładowa implementacja stogowej kolejki priorytetowej

```
// c.d.:
  private void sink(int index) {   // zatapianie (opuszczanie) elementu
     boolean isDone=false;
     int child;
     while(!isDone && (child=2*index+ 1 ) < _list.size()) {
       if (child < _list.size()-1 &&
            _comparator.compare(_list.get(child), _list.get(child+1)) < 0)
                ++child;
       if (_comparator.compare(_list.get(index), _list.get(child)) < 0)
                swap(index, child);
         else isDone=true;
       index = child;
     }
  }
  public void clear() {
     _list.clear();
  }
  public int size() {
     return _list.size();
  }
  public boolean isEmpty() {
     return _list.isEmpty();
  }
}
```

Kolejki priorytetowe

Kolejki z ograniczonym, niewielkim zbiorem wartości priorytetów realizuje się z użyciem tablicy (lub listy) zwykłych kolejek. Każda kolejka odpowiada jednej wartości priorytetu i zachowuje kolejność dołączania elementów do kolejki.

Podsumowanie

Kolejki priorytetowe zrealizowane z użyciem listy nieuporządkowanej i listy uporządkowanej są bardzo mało efektywne (dla przeciętnych przypadków) w porównaniu do kolejki stogowej.

Złożoność obliczeniowa kolejki stogowej jest logarytmiczna (jest to cecha charakterystyczna algorytmów opartych na drzewach binarnych).

W najbardziej korzystnym przypadku (dodawanie do kolejki elementów posortowanych w kolejności rosnącej) realizacja kolejki priorytetowej opartej na liście uporządkowanej daje bardzo dobre wyniki. Przyczyna jest oczywista...