W. 4.3
Kolejka
(Queue)

Idea

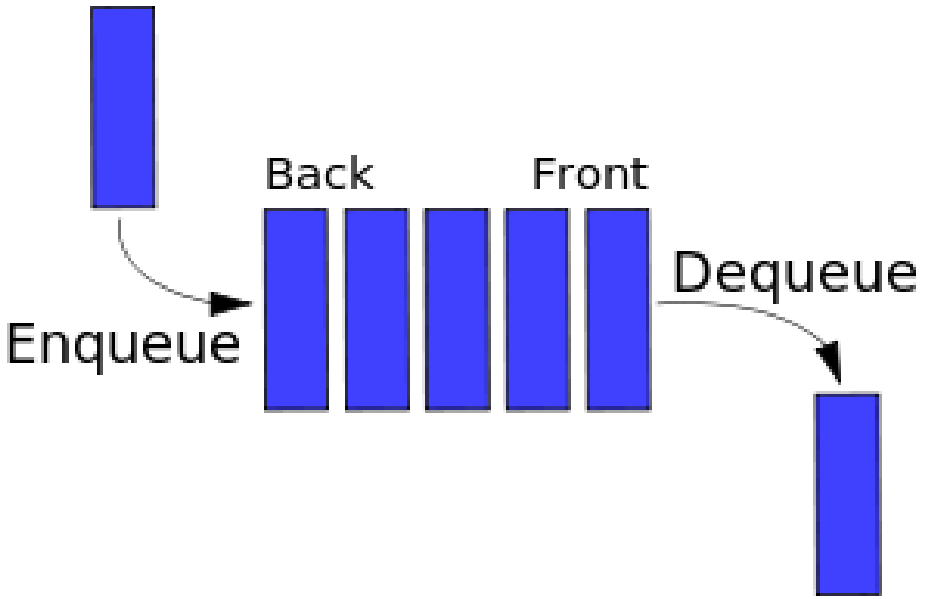
- Mamy uszeregowane elementy (jak w kolejce do sklepu).
- Zadajemy sposób dodawania i zdejmowania elementów z kolejki:
  - Enqueue - Elementy do kolejki dodajemy w określony sposób.
  - Dequeue - Elementy z kolejki zdejmujemy w określony sposób.
- Te operacje determinują typ kolejki.

Przykłady kolejek jako ASD

- **FIFO (First In, First Out), bufor** – kolejka ta przechowuje elementy do czasu gdy będą mogły być obsłużone. Elementy zdejmowane są w takiej samej kolejności w jakiej zostały włożone.
- **FILO (First In, Last Out), stos** - pierwszy wstawiony element opuszcza kolejkę jako ostatni. Struktura przypomina stos książek – książka może być wyjęta tylko, gdy wszystkie książki nad nią zostały już zdjęte.
- **Kolejka priorytetowa** – kolejka w której elementy są zdejmowane zgodnie z (największym) priorytetem, np. ważna osoba, znajomy.
  - Zauważ, że kolejkę FIFO i FILO możemy zaimplementować jako kolejkę priorytetowe:
    - FIFO – element włożony wcześniej ma większy priorytet niż ten włożony później;
    - FILO - największy priorytet ma element włożony jako ostatni – jest to odwrócenie priorytetów w stosunku do FIFO;
- Mamy dodatkowe wariacje nazw (proszę przeanalizować działanie):
  - LILO (Last In, Last Out) = FIFO, bufor
  - LIFO (Last In, First Out) = FILO, stos

FIFO/bufor
Implementacja

Kolejka FIFO i jej implementacja



- Najłatwiejszym sposobem implementacji jest wykorzystanie listy wraz z:
  - dodawaniem elementu na początek listy;
  - zdejmowanie elementu z końca listy;
- Proszę zaimplementować kolejkę w C++ przy użyciu listy jedno i dwukierunkowej.
  - Czy użycie listy dwukierunkowej daje przewagę nad listą jednokierunkową? (Czy potrzebujemy wskaźnika do węzła poprzedniego?)

FIFO na listach – Python globalna zmienna

```
queue = []

def length():
    """Returns number of waiting customers"""
    return len(queue)

def show():
    """print list of customers, longest waiting customer at end."""
    for name in queue:
        print("waiting customer: {}".format(name))

def add(name):
    """Customer with name 'name' joining the queue"""
    queue.insert(0, name)

def next():
    """Returns name of next to serve, removes customer from queue"""
    return queue.pop()


add('Spearing'); add('Fangohr'); add('Takeda')
show(); next()
```

- To podejście wykorzystuje globalną listę **queue**, co nie jest dobrym podejściem.
- Możemy operować tylko na jednej kolejce.

FIFO na listach – Python lokalne zmienne

```
def length(queue):
    return len(queue)

def show(queue):
    for name in queue:
        print("waiting customer: {}".format(name))

def add(queue, name):
    queue.insert(0, name)

def next(queue):
    return queue.pop()


q1 = []
q2 = []
add(q1, 'Spearing'); add(q1, 'Fangohr'); add(q1, 'Takeda')
add(q2, 'John'); add(q2, 'Peter')
print("{} customers in queue1:".format(length(q1))); show(q1)
print("{} customers in queue2:".format(length(q2))); show(q2)
```

- To podejście używa kolejek przekazywanych jako argumenty do funkcji.
- Dzięki temu możemy operować na większej liczbie kolejek.
- Problemem jest to, że struktura przechowująca kolejkę (lista) oraz funkcje operujące na niej są niepowiązane, a w prawidłowym podejściu i rzeczywistości powinny być.

FIFO na listach – Python podejście obiektowe

```
class Fifoqueue:
    def __init__(self):
        self.queue = []

    def length(self):
        return len(self.queue)

    def show(self):
        for name in self.queue:
            print("waiting customer: {}".format(name))

    def add(self, name):
        self.queue.insert(0, name)

    def next(self):
        return self.queue.pop()

q1 = Fifoqueue(); q2 = Fifoqueue()
q1.add('Spearing'); q1.add('Fangohr'); q1.add('Takeda')
q2.add('John'); q2.add('Peter')
print("{} customers in queue1:".format(q1.length())); q1.show()
```

- W tym podejściu dane i funkcje które na nich operują są powiązane (enkapsulacja) w strukturze klasy.

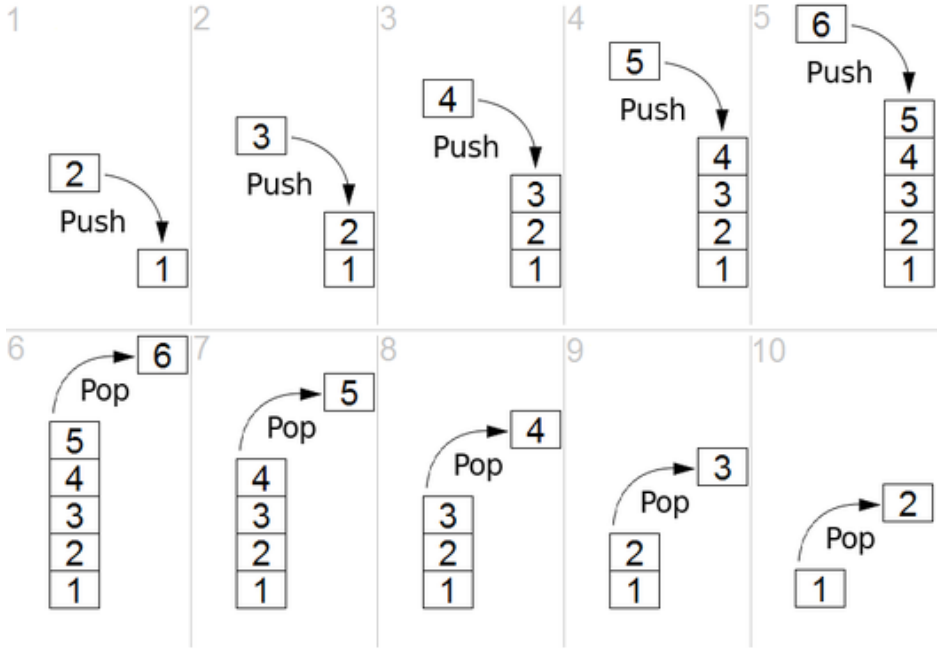
FIFO na listach – Python podejście funkcyjne

```
def make_queue():
    queue = []

    def length():
        return len(queue)

    def show():
        for name in queue: print("waiting customer: {}".format(name))

    def add(name):
        queue.insert(0, name)

    def next():
        return queue.pop()

    return add, next, show, length


q1_add, q1_next, q1_show, q1_length = make_queue()
q2_add, q2_next, q2_show, q2_length = make_queue()
q1_add('Spearing'); q1_add('Fangohr'); q1_add('Takeda')
q2_add('John'); q2_add('Peter')
print("{} customers in queue1:".format(q1_length())); q1_show()
print("{} customers in queue2:".format(q2_length())); q2_show()
```

- W podejściu funkcyjnym funkcja (wyższego rzędu) **make_queue()** produkuje kolejkę zestaw funkcji na niej operujących:
  - return **add, next, show, length**
- W tym podejściu kolejka **queue** zostaje ‘zaszyta’ w tak utworzonych funkcjach operujących na kolejce. Taka technika nazywa się ‘zapamiętywaniem’ (memorization).

FILO/stos (stack)
implementacja

Kolejka FILO/LIFO/stos i jej implementacja



- Należy zaimplementować:
  - Push – dodawanie elementu na szczyt stosu;
  - Pop – zdejmowanie (i zwracanie) elementu ze szczytu stosu;
- Proszę zaimplementować stos przy pomocy listy jedno i dwukierunkowej w języku C++.
  - Proszę się zastanowić, czy wykorzystanie listy dwukierunkowej przynosi korzyść w stosunku do użycia listy jednokierunkowej?

Stos - python

```
stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() fucntion to pop
# element from stack in
# LIFO order
print('\nElements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are poped:')
print(stack)
```

- Ta implementacja wykorzystuje listy w Pythonie oraz ich metody
  - Append – dodaj na koniec listy
  - Pop – zdejmij element z końca i zwróć go;
- Zauważ, że koniec listy jest wierzchołkiem stosu !!!
- Zaimplementuj stos w podejściu
  - Obiektowym – napisz klasę opakowującą ten kod;
  - Funkcyjnym – napisz funkcję generującą kolejkę, t.j. zwracającą metody push i pop z „zaszytą” kolejką.

Zastosowanie stosu
Kalkulator dla notacji Polskiej (prefiksowej)

Sposoby zapisu wyrażeń arytmetycznych

- Notacja infiksowa (operator w środku):
  - 2+3*4
- Notacja prefiksowa/Polska (operator przed argumentami):
  - +2*34
- Notacja postfiksowa/odrwotna notacja Polska (operator za arhumentami):
  - 234*+
- Notacja prefiksowa jest stosowana w językach funkcyjnych rodziny LISP:
  - (+ 2 (* 3 4))
- Zauważ, że w notacji pre- i postfixowej nie musisz stosować nawiasów do grupowania – wystarczy priorytet operatorów. Przykłady:
  - (A + B) * (C + D) - infix
  - * + A B + C D - prefix
  - A B + C D + * - postfix
- Wyrażenia w notacjach pre- i postfiksowych są łatwiejsze do przetworzenia i obliczenia. Istotną rolę gra stos.

Ewaluacja wyrażenia w notacji Polskiej

```
def is_operand(c):
    return c.isdigit()

def evaluate(expression):
    stack = []

    for c in expression[::-1]:
        # push operand to stack
        if is_operand(c):
            stack.append(int(c))
        else:
        # pop values from stack can calculate the result
        # push the result onto the stack again
            o1 = stack.pop()
            o2 = stack.pop()
            if c == '+':
                stack.append(o1 + o2)
            elif c == '-':
                stack.append(o1 - o2)
            elif c == '*':
                stack.append(o1 * o2)
            elif c == '/':
                stack.append(o1 / o2)
    return stack.pop()
# Driver code
if __name__ == "__main__":
    test_expression = "+9*26"
    print(evaluate(test_expression))
```

- Funkcja is_operand() sprawdza, czy argument jest liczbą (a nie operatorem).
- evaluate() - oblicza argument będący wyrażeniem w notacji Polskiej:
  - W pętli for po literach w wyrażeniu (string):
    - Pobierz znak z wyrażenia;
    - Jeżeli to jest to cyfra to włóż na stos (po konwersji na int)
    - Jeżeli jest to operator dwuargumentowy (+,*, /) to pobierz dwie liczby ze stosu, oblicz wartość i włóż wynik na stos.
  - Po przetworzeniu wynik końcowy jest na stosie.

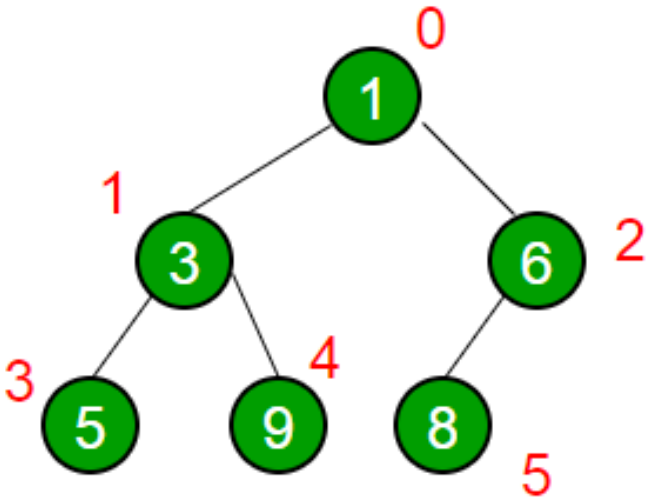
Kolejka priorytetowa

Implementacja

- Z kolejki priorytetowej możemy wyjmować elementy wg określonego priorytetu, np. największa liczba.
- Najprostszą implementacją jest nieposortowana lista w której wyjmujemy największe elementy. Ta implementacja nie jest efektywna.
- Efektywnym podejściem jest użycie sterty/kopca (heap).
- Zaimplementujemy kolejkę priorytetową przy użyciu binarnego kopca.

Binarny kopiec - Implementacja

Kopiec binarny



1	3	6	5	9	8
0	1	2	3	4	5

- Własność:
  - Jest to drzewo (odwrócone) w którym brakuje co najwyżej ostatnich elementów – gałęzie mają porównywalną długość.
  - **Element w węźle jest od elementów potomnych:**
    - **Większy – maksymalny kopiec (max heap).**
    - **Mniejszy – minimalny kopiec (min heap).**
- Implementacja przy pomocy listy:
  - Arr[0] – korzeń (root);
  - Arr[(i-1)/2] – węzeł (node);
  - Arr[(2*i)+1] – lewy węzeł potomny (left node);
  - Arr[(2*i)+2] – prawy węzeł potomny (right node);

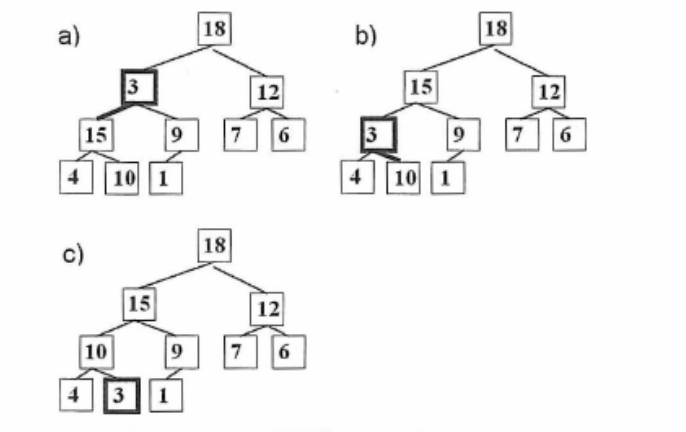
Klasa minimalnego kopca – Min Heap

- class **MinHeap**
- { int *harr; // pointer to array of elements in heap
- int capacity; // maximum possible size of min heap
- int heap_size; // Current number of elements in min heap
- public:
- // Constructor
- MinHeap(int capacity);
-
- // to heapify a subtree with the root at given index
- void MinHeapify(int);
-
- int parent(int i) { return (i-1)/2; }
-
- // to get index of left child of node at index i
- int left(int i) { return (2*i + 1); }
-
- // to get index of right child of node at index i
- int right(int i) { return (2*i + 2); } ;

- // to extract the root which is the minimum element
- int **extractMin();**
-
- // Decreases key value of key at index i to new_val
- void **decreaseKey(int i, int new_val);**
-
- // Returns the minimum key (key at root) from min heap
- int **getMin()** { return harr[0]; }
-
- // Deletes a key stored at index i
- void **deleteKey(int i);**
-
- // Inserts a new key 'k'
- void **insertKey(int k);**
- }
-
- // Constructor: Builds a heap from a given array a[] of given size
- MinHeap::MinHeap(int cap)
- { heap_size = 0;
- capacity = cap;
- harr = new int[cap]; }
-
- void swap(int *x, int *y) { int temp = *x; *x = *y; *y = temp; }

„Kopcowanie” - heapify

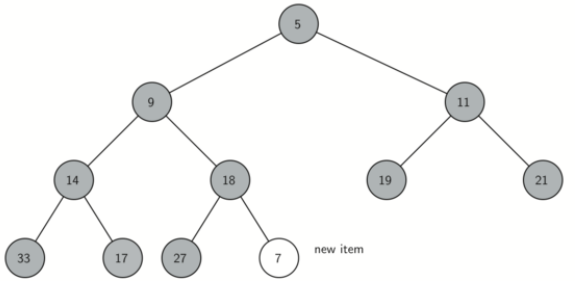
- void MinHeap::**MinHeapify**(int i)
- { int l = left(i);
- int r = right(i);
- int smallest = i;
- if (l < heap_size && harr[l] < harr[i])
- smallest = l;
- if (r < heap_size && harr[r] < harr[smallest])
- smallest = r;
- if (smallest != i)
- { swap(&harr[i], &harr[smallest]);
- **MinHeapify**(smallest); } }

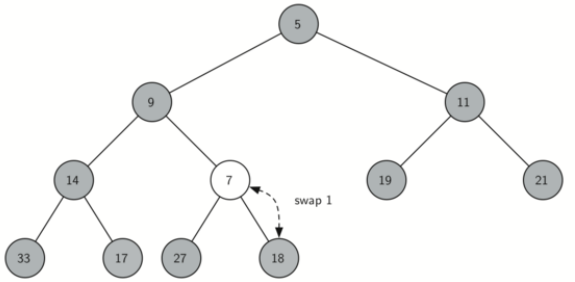


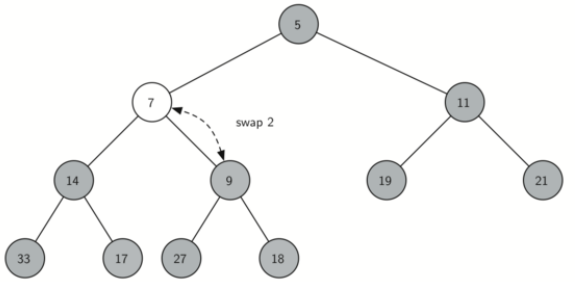
- Przywracanie własności kopca rozpoczyna się od węzła o indeksie **i** w sposób rekurencyjny.
- Tworzenie całego kopca (od węzła i=0) jest klasy O(log(n)).

Wstawianie elementu

- void MinHeap::insertKey(int k)
- { if (heap_size == capacity)
- { cout << "\nOverflow: Could not insertKey\n";
- return; }
-
- // First insert the new key at the end
- heap_size++;
- int i = heap_size - 1;
- harr[i] = k;
-
- // Fix the min heap property if it is violated
- while (i != 0 && harr[parent(i)] > harr[i])
- { swap(&harr[i], &harr[parent(i)]); i = parent(i); } }







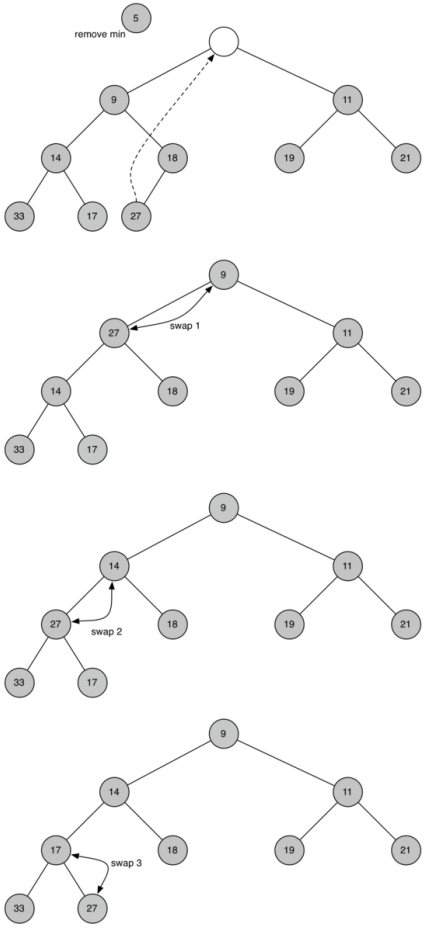
Zmniejsz wartość w wybranym węźle

- void MinHeap::decreaseKey(int i, int new_val)
- { harr[i] = new_val;
- while (i != 0 && harr[parent(i)] > harr[i])
- { swap(&harr[i], &harr[parent(i)]); i = parent(i); }
- }

- Ustawiamy nową wartość **new_val** w węźle o indeksie **i**.
- Przesuwamy wstawiony element w górę kopca dopóki własność kopca nie będzie spełniona.

Usuwamy element najmniejszy

- int MinHeap::extractMin()
- { if (heap_size <= 0) return INT_MAX;
- if (heap_size == 1)
- { heap_size--; return harr[0]; }
-
- // Store the minimum value, and remove it from heap
- int root = harr[0];
- harr[0] = harr[heap_size-1];
- heap_size--;
- **MinHeapify**(0);
- return root; }



Usuwamy element o indeksie i

- void MinHeap::deleteKey(int i)
- { decreaseKey(i, INT_MIN); extractMin(); }

Sortowanie przez kopcowanie
(Heap sort)

Algorytm

- Algorytm
  - Pobierz tablicę T o długości n do posortowania.
  - L=[]
  - Stwórz kopiec minimalny z tablicy T
  - Dla i z [0..n]:
    - T.MinHeapify(i)
  - Tablica T jest posortowana.
- Złożoność obliczeniowa: min=max=średnia = O(n *log(n)).
- Działanie:
  - https://www.youtube.com/watch?v=Xw2D9aJRBY4

Implementacja

- void **heapify(int arr[], int n, int i)**
- { int largest = i; // Initialize largest as root
- int l = 2*i + 1; // left = 2*i + 1
- int r = 2*i + 2; // right = 2*i + 2
- if (l < n && arr[l] > arr[largest])
- largest = l;
- if (r < n && arr[r] > arr[largest])
- largest = r;
- if (largest != i)
- { swap(arr[i], arr[largest]);
- heapify(arr, n, largest); } }

- void **heapSort(int arr[], int n)**
- { for (int i = n / 2 - 1; i >= 0; i--)
- heapify(arr, n, i);
- for (int i=n-1; i>0; i--)
- { swap(arr[0], arr[i]); heapify(arr, i, 0); } }
-
- // Driver program
- int main()
- { int arr[] = {12, 11, 13, 5, 6, 7};
- int n = sizeof(arr)/sizeof(arr[0]);
- heapSort(arr, n); }

Literatura dodatkowa

- https://www.geeksforgeeks.org/doubly-linked-list/
- Rozdział 5 - Algorytmy. Struktury danych i techniki programowania.
- Rozdziały 10 – T. Cormen, ‘Wprowadzenie do algorytmów’, PWN
- Rozdział 6.2, 2 - Data Structures and Algorithms using Python.

Koniec