Sprawozdanie
AISDI, **kopce**

wykonujący:
Jakub Rozkosz
Hubert Gołębiowski

Podział pracy:
Jakub Rozkosz:

- implementacja kopca binarnego
- napisanie funkcji do rysowania wykresów
- wykonanie pomiarów
- wygenerowanie wykresów
- pisanie sprawozdania

Hubert Gołębiowski:

- implementacja kopcja n-arnego
- napisanie funkcji print()
- pisanie sprawozdania

Pliki:
binary_heap.py - plik z implementacją kopca rosnącego binarnego (kod stricte pod dwójkę "dzieci"). Zawiera klasę BinaryHeap, która posiada pole 'heap' - kopiec w postaci listy. Metodami klasy są: insert_value(), która korzysta z metody _up_heap() (wstawianie wartości do kopca)
delete_root(), która korzysta z metody _down_heap() (usuwanie korzenia kopca)
print() - wyświetlanie kopca w postaci drzewa.

n_ary_heap.py - plik z implementacja kopca n-arnego. Zawiera klasę NHeap. W konstruktorze podajemy liczbę n, która odpowiada ilości dzieci którą posiada każdy węzeł. Kopiec rośnie w dół, co oznacza, że na pierwszej pozycji jest najmniejsza wartość. Klasa posiada metody: insert_value() (do wstawiania wartości do kopca), delete_root() (do usunięcia i pobrania najmniejszej wartości z kopca). Używają one pomocniczych metod _upHeap() oraz _downHeap(). Oprócz tego posiada ona jeszcze metodę print() do wyświetlania kopca.

test_n_ary.py - plik do sprawdzania poprawnosci działania kopca n-arnego. Plik wyświetla w terminalu kopiec do którego dodawane są nowe wartości, a następnie usuwane.
test_binary.py - plik, w którym testowana jest poprawnośc dodawania elementów do kopca binarnego, usuwania korzenia oraz wyświetlania w postaci drzewa.

drawing_plots.py - plik z funkcją do rysowania wykresów na podstawie pomiarów czasowych.
time_measure.py - plik z funkcją do wykonywania pomiarów czasowych, w argumencie można podać ilość repetycji pomiarów, zwracana jest średnia z wszystkich pomiarów.
main.py - plik z zaimportowanymi, powyższymi modułami, w którym wykonywane są pomiary oraz generowane wykresy.

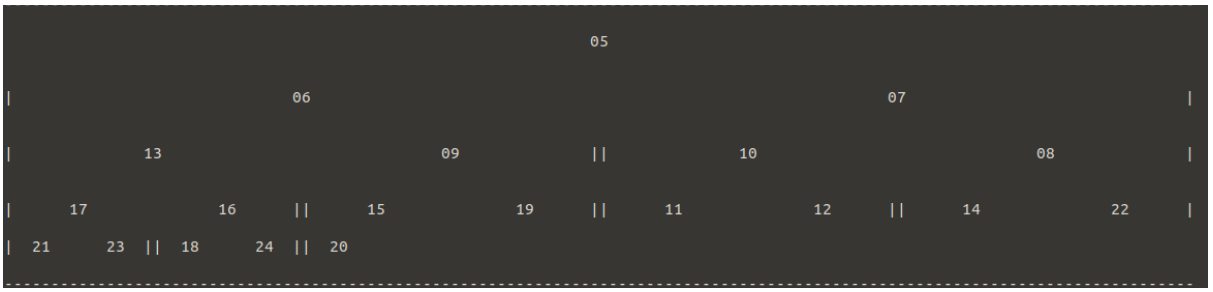
Instrukcja:
Aby przetestować kopiec binarny należy uruchomić plik ***test_binary.py***

Aby przetestować kopiec n-arny należy w pliku ***test_n_ary.py*** wybrać krotność kopca wpisując ją jako argument w konstruktorze obiektu klasy NHeap, a następnie uruchomić program.

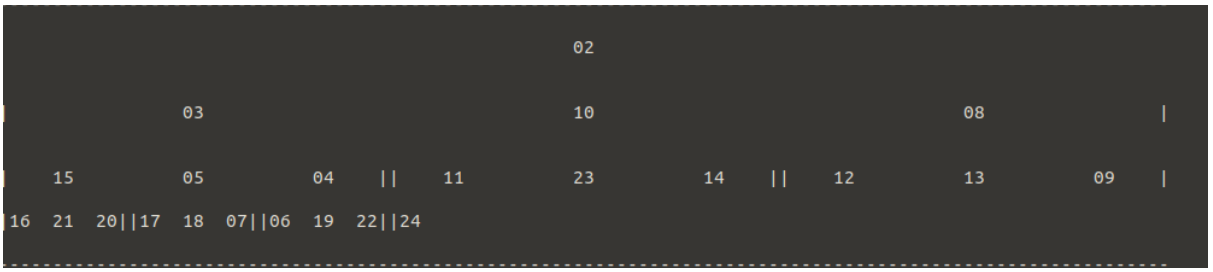
Opis:
W naszym projekcie porównaliśmy ze sobą czasy wykonywania poszczególnych metod dla czterech rodzajów kopców (binarny, 2-arny, 3-arny, 4-arny). Kopiec binarny został sprawdzony w dwóch formach - jako instancja binary heap oraz jako instancja n-ary heap o argumencie 2. Do pomiarów wykorzystaliśmy funkcje process_time() z biblioteki time. Pomiary wykonywaliśmy dla losowo wygenerowanych danych. Ich ilość była z przedziału od 10.000 do 100.000. Najpierw wszystkie elementy listy wstawiliśmy do kopca. Następnie po kolei usuwaliśmy korzenie z powstałego kopca do momentu jego całkowitego usunięcia. Warto dodać, że wszystkie kopce otrzymywały dokładnie taką samą listę argumentów. Każdy proces powtarzaliśmy 10 razy i wyciągaliśmy średnią czasową.

Wyświetlanie kopców:

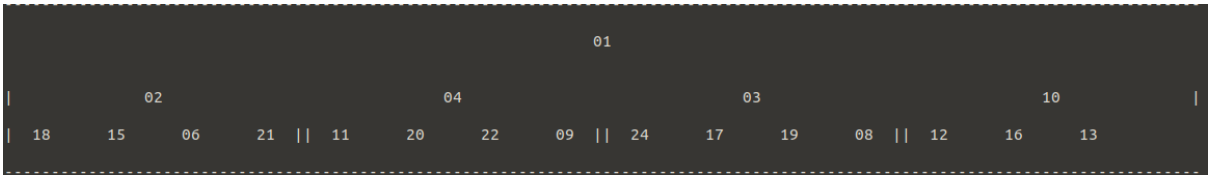
2-arny



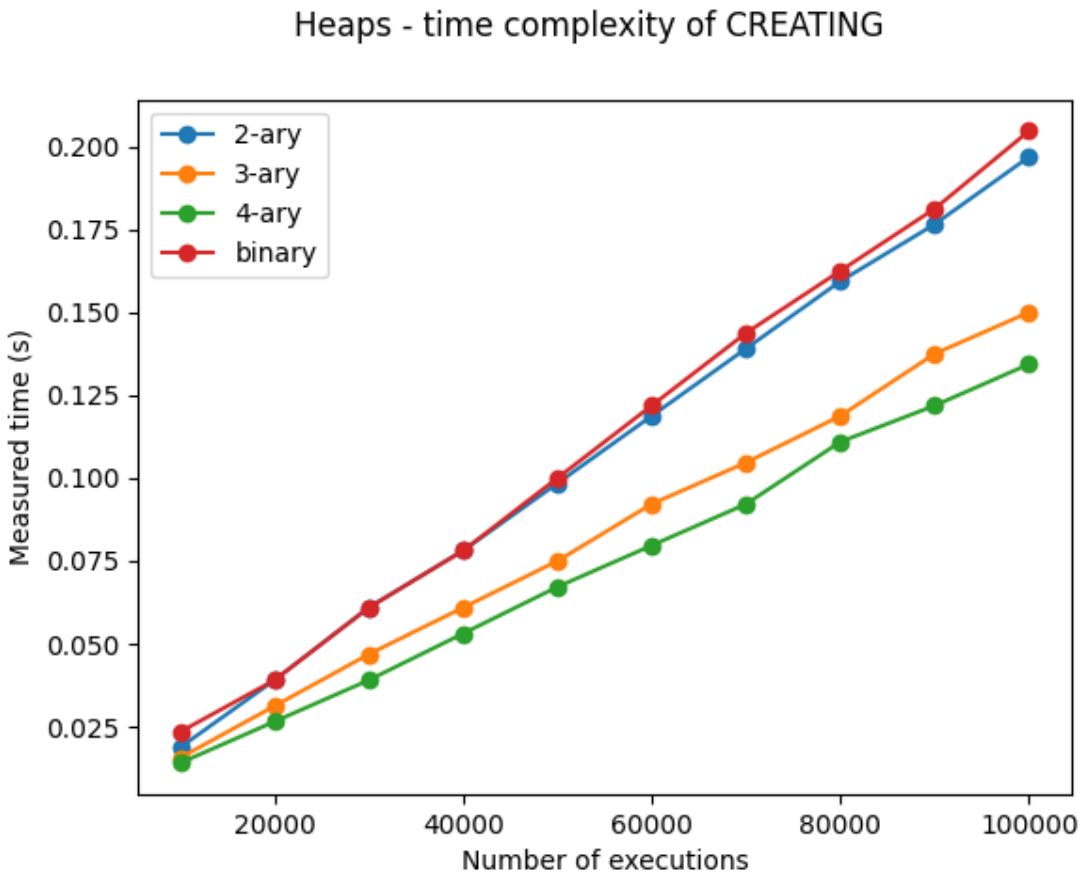
3-arny



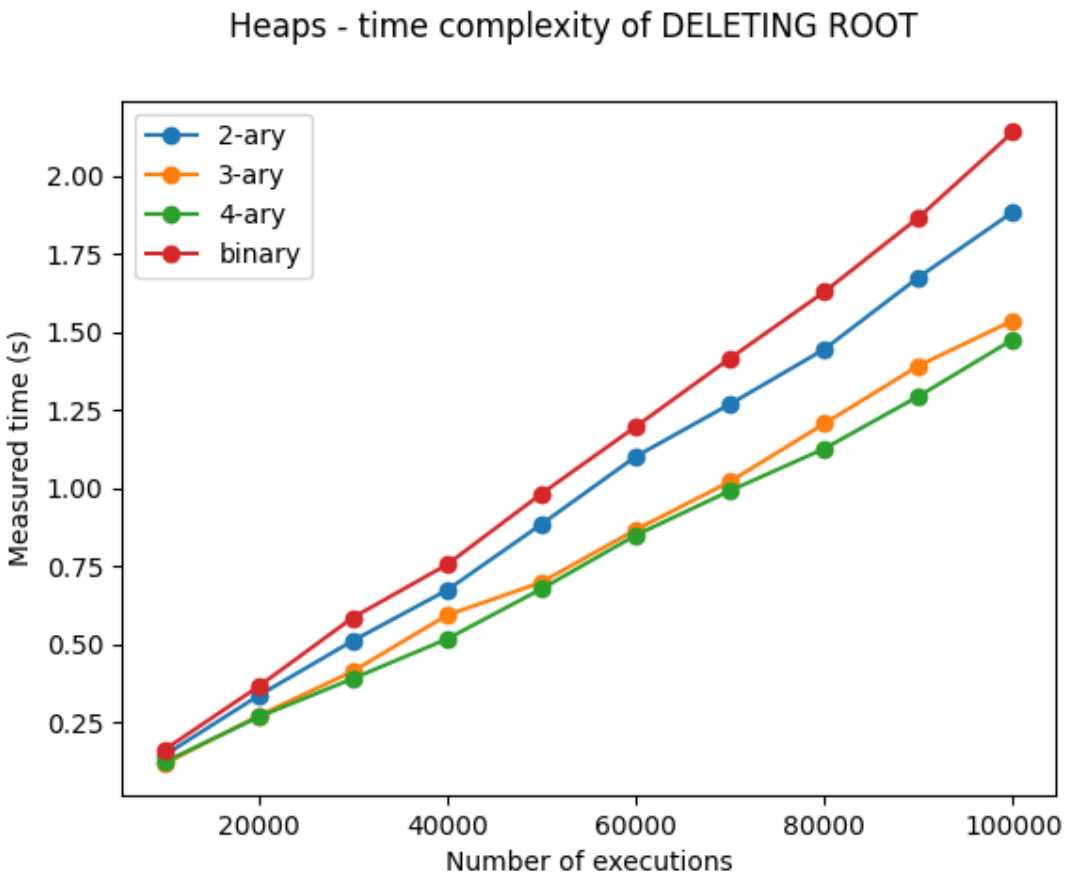
4-arny



Wyniki:



Jak możemy zauważyć złożoność czasowa kopca binarnego bardzo przypomina wykres funkcji liniowej (złożoność n). Mniejsza ilość dzieci powoduje większą wysokość. A więc przy wstawianiu elementu to kopca wartość musi pokonać więcej poziomów w górę - funkcja *up_heap* jest kosztowniejsza. Tworzenie kopców 3-arnego oraz 4-arnego zajmuje już trochę mniej czasu. 4-narny okazał się najszybszy.
Więcej dzieci powoduje zmniejszenie wysokości. Większa liczba dzieci natomiast skutkuje kosztowniejszym *down_heapem,* aczkolwiek przy naszych krotnościach kopców nie było to jeszcze odczuwalne.



Wykres praktycznie identyczny jak poprzedni. Najszybszy jest kopiec 4-arny, następnie 3-arny a najwolniejsze są kopce binarne. Może się to wydawać nieintuicyjne, gdyż funkcja *down_heap*, która jest używana przy usuwaniu korzenia, dla kopców o większej liczbie dzieci powinna być wolniejsza (kosztowniejsze wyszukiwanie najmniejszego dziecka). I faktycznie tak jest, przy czym większa ilość dzieci oznacza, że kopiec jest niższy, co rekompensuje dłuższe poszukiwanie najmniejszego dziecka w funkcji *down_heap*.

Wnioski:

Wśród rozpatrywanych przez nas kopców kluczowy wpływ na złożoność czasową miała ich wysokość. 4-arny przy tworzeniu oraz usuwaniu korzenia okazał się najszybszym kopcem. Natomiast mamy świadomość, że zależność: ‘im bardziej-narny kopiec tym szybszy’ nie jest w pełni prawdziwa, gdyż większa ilość dzieci również niesie za sobą kosztowność czasową.