Raport z laboratorium nr: ***A3***

Imię i nazwisko studenta: ***Szymon Bobrowski***

nr indeksu: ***401533***

1. Kod źródłowy

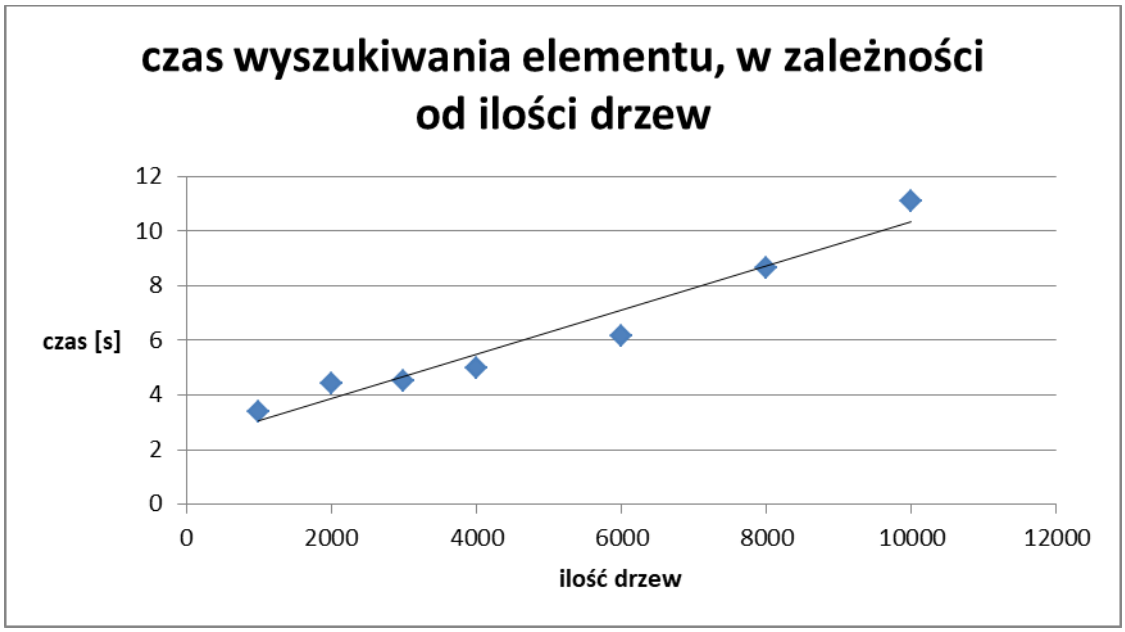
```
1.  def insert(root,node):
2.      if root is None:
3.          root = node
4.      else:
5.          if root.val < node.val:
6.              if root.right is None:
7.                  root.right = node
8.              else:
9.                  insert(root.right, node)
10.         else:
11.             if root.left is None:
12.                 root.left = node
13.             else:
14.                 insert(root.left, node)
```

Wybrałem powyższy fragment kodu, ponieważ jest on podstawą całego programu i to od jego implementacji zależą wszystkie funkcje i postać naszej struktury. Był to także fragment, którego zrozumienie i implementacja sprawiły mi najwięcej trudności. Binarne drzewa poszukiwań są kolejnym świetnym przykładem, w którym rekurencja bierze górę nad funkcjami iteracyjnymi. Nie inaczej jest także i w tym przypadku. W porównaniu do swojej iteracyjnej odpowiedniczki funkcja rekurencyjna jest znacznie bardziej zwięzła i klarowna. Ten fragment w znacznym stopniu pomógł mi zrozumieć rekurencję i przekonał do jej stosowania w praktyce.

2. Wyniki i podsumowanie

Dla utworzonej przeze mnie struktury, zbadam teraz szybkość wykonywania poszczególnych elementów. Za każdym razem przyjmuję drzewo binarne poszukiwań o dokładnie 40 węzłach. Wartością, która się zmienia jest ilość tworzonych drzew. Zebrane wyniki prezentuję w poniższej tabeli oraz na wykresie.

ilość drzew	minimum[s]	maximum[s]	insert[s]	search[s]
1000	1,47	1,44	2,13	3,41
2000	1,47	1,51	1,74	4,44
3000	1,47	1,48	1,46	4,5
4000	1,49	1,56	1,44	5
6000	1,5	1,49	1,34	6,15
8000	1,5	1,5	1,32	8,65
10000	1,51	1,51	1,141	11,09



Jak widać jedynie czas wyszukiwania elementu w strukturze zależy od ilości drzew. W przypadku pozostałych operacji wystarczy brać pod uwagę tylko długość pojedynczego drzewa. Dzieje się tak dlatego, ponieważ we wszystkich operacjach oprócz SEARCH(), wiemy, na którym konkretnym drzewie będziemy operować. Funkcje min. oraz max. bazują na metodzie inorder traversal, co oznacza, że algorytm przechodzi przez wszystkie węzły drzewa. Zatem w obu przypadkach złożoność obliczeniowa wynosi O(n). Nie jest to rozwiązanie optymalne, jednak jego zaletą jest łatwość implementacji. Przewagę drzew bst (a tym samym naszej struktury) nad tablicami można zaobserwować w przypadku dodawania elementu. Funkcja INSERT() przechodzi po jednym wierzchołku na dany poziom co daje nam złożoność O(h), gdzie h jest wysokością drzewa. Jedynie w najgorszym przypadku h=n ta złożoność wynosi O(n). Jeśli chodzi, o metodę SEARCH() trzeba tutaj uwzględnić zarówno ilość wierzchołków drzewa jak i ilość wszystkich drzew. Ponieważ ta funkcja także opiera się na inorder traversal, odwiedzamy każdorazowo wszystkie wierzchołki. Jej złożoność można zatem oszacować jako O(m*n), gdzie m jest liczbą drzew w strukturze.

Przykładem zastosowania tej struktury może być np. stworzenie prostego algorytmu sortującego dla kilku ciągów liczb. Przechowując dane w ten sposób możemy wykorzystać nieskomplikowaną metodę inorder, aby otrzymać posortowane wartości, co daje nam algorytm o złożoności O(n).