1. Napisać rekurencyjną funkcję `wsk dodaj(wsk T, int y)`, która dodaje do binarnego *uporządkowanego* drzewa wskazywanego przez `T` nową wartość `int y`, zgodnie z uporządkowaniem. Podać także drugą wersję funkcji, w której wskaźnik na drzewo przekazywany jest przez parametr typu adres wskaźnika, `void dodaj(wsk *T, int y)`.

2. Podobnie jak w zadaniu 1, tym razem dodajemy element do binarnego drzewa *nieuporządkowanego*. Wybór lewego lub prawego poddrzewa, do którego w kolejnym kroku rekurencji dodany będzie następny element odbywa się losowo.

3. Napisać funkcję `wsk szukaj(wsk T, int y)`, która sprawdza czy element o wartości `y` znajduje się w a) uporządkowanym, b) nieuporządkowanym drzewie binarnym wskazywanym przez `T`. Funkcja powinna zwracać wskaźnik na węzeł drzewa zawierający `y` lub `NULL`, jeśli go nie znaleziono. Druga wersja rozwiązania powinna zwracać adres wskaźnika na węzeł zawierający `y` lub `NULL`.
UWAGA: W przypadku drzewa uporządkowanego dobre rozwiązanie zwraca adres wskaźnika o wartości `NULL` w miejscu, gdzie zgodnie z porządkiem powinien znaleźć się element `y`, jeśli nie ma go dotąd w drzewie.

4. Drukowanie zawartości drzewa `T` w porządku a) prostym i b) odwrotnym.

5. Jeśli liczba elementów `n`, z których należy zbudować drzewo jest znana z góry, podać rozwiązanie rekurencyjne `wsk utworz(int n)` (w drugiej wersji `void utworz(wsk *T, int n)`), które tworzy z nich drzewo zapisując pierwszy element w korzeniu drzewa, a pozostałych `n-1` elementów rozdziela po połowie między lewe i prawe poddrzewa. Funkcja zwraca jako wartość wskaźnik na korzeń drzewa (w drugiej wersji — przez parametr `*T`).

6. Napisać funkcję, która zlicza ile razy wartość `x` występuje w drzewie a) uporządkowanym, b) nieuporządkowanym.

7. a) Napisać funkcję `wsk nast(wsk T)`, która zwraca wskaźnik (w drugiej wersji — adres tego wskaźnika) na element będący następnikiem `T->x` w uporządkowanym drzewie o korzeniu `T` lub `NULL`, gdy tego następnika nie ma. b) Podobie jak w a), tym razem dla poprzednika wartości `T->x`.

8. Napisać funkcję `wsk usun(wsk T, int y)`, która usuwa z a) nieuporządkowanego, b) uporządkowanego drzewa `T` pierwszy napotkany element o wartości `y`. Funkcja powinna zwracać wskaźnik na wynikowe drzewo, które może być puste. Druga wersja rozwiązania powinna wykorzystać przekazywanie wynikowego `T` przez odp. parametr.
UWAGA: Usunięcie wartości `y` nie może zepsuć struktury drzewa uporządkowanego w zadaniu b).

9. Podobnie jak w zadaniu 8, lecz obecnie funkcja powinna usuwać *wszystkie* wystąpienia `y` w `T`.

10. Napisać funkcję `int lw(wsk T)`, która zwraca jako wartość liczbę węzłów w drzewie `T`.

11. Napisać funkcję `int ll(wsk T)`, która zwraca jako wartość liczbę liści (węzłów końcowych) w drzewie `T`.

12. Napisać funkcję `int wys(wsk T)`, która zwraca jako wartość wysokość (liczbę poziomów) drzewa `T`.

13. Napisać funkcję `void level(wsk T, int k)`, która wypisuje wartości wszystkich węzłów drzewa `T` na poziomie `k`, od lewej do prawej. Korzeń drzewa ma poziom 1.

14. Napisać funkcję `int szer(wsk T)`, która zwraca jako wartość szerokość (maksymalną liczbę węzłów na jednym poziomie) drzewa `T`.

15. Napisać funkcję, która oblicza *wyważenie* drzewa, jako maksymalną wśród wszystkich jego węzłów wartość różnicy między a) wysokościami, b) liczebnościami lewego i prawego poddrzewa.

16. Napisać funkcję typu logicznego, która porównuje dwa drzewa `T1` i `T2`, zwracając wartość 1, gdy drzewa są identyczne pod względem struktury i zawartości lub 0 w przeciwnym wypadku.

17. Uogólnić funkcje `nast` i `poprz` z zad. 7, tak by zwracały wskaźniki na następnik i poprzednik *dowolnego* wskazanego elementu `v` w uporządkowanym drzewie `T`. Należy zwrócić uwagę, że zadanie komplikuje się nieco, gdy np. przy poszukiwaniu następnika `v` węzeł ten nie posiada prawego poddrzewa. Wówczas następnik `v` może znajdować się wyżej w drzewie: jest to taki element `w`, dla którego `v` jest poprzednikiem. Do prawidłowego rozwiązania tego zadania potrzeba, by każdy węzeł drzewa posiadał wskaźnik na swojego rodzica.

18. Napisać funkcję logiczną, która porównuje 2 uporządkowane drzewa pod względem ich *zawartości*. Struktura drzew może być różna. Należy wykorzystać funkcje z zadania 17.

19. Na ćwiczeniach omówimy tzw. kodowanie Huffmanna stosowane w kompresji danych. Napisać procedurę, która na podstawie danych w postaci par "*znak : liczba wystąpień*" tworzy drzewo Huffmanna i następnie drukuje tablicę kodów dla zadanego zestawu znaków.