LISTA 1 - JA

ZAD. 1 NAPISZ REKURENCYJNE FUNKCJE, KTÓRE DLA DANEGO DRZEWA BINARNEGO T OBLICZAJĄ:

- ILOŚĆ WIERZCHOŁKÓW W T

```
int policz_wezly(Node *n)
{
    if(n == NULL)
        return 0;
    return policz_wezly(n->left)+policz_wezly(n->right)+1;
}
```

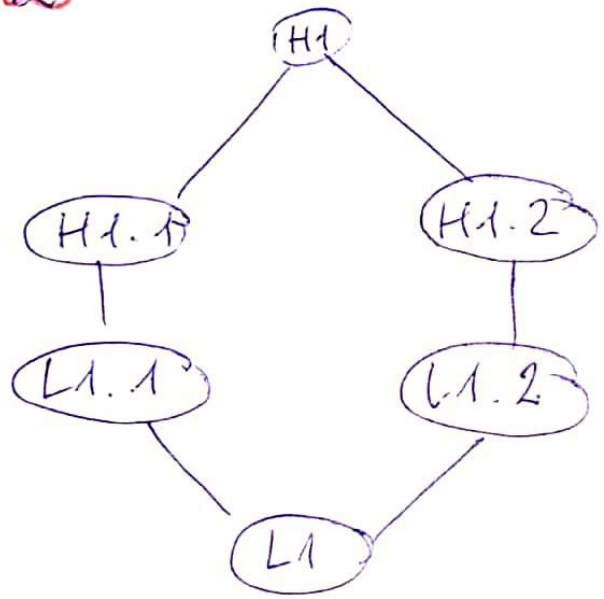
```
int liczba_wierzcholkow(Tree T)
{
  return policz_wezly(T.root);
}
```

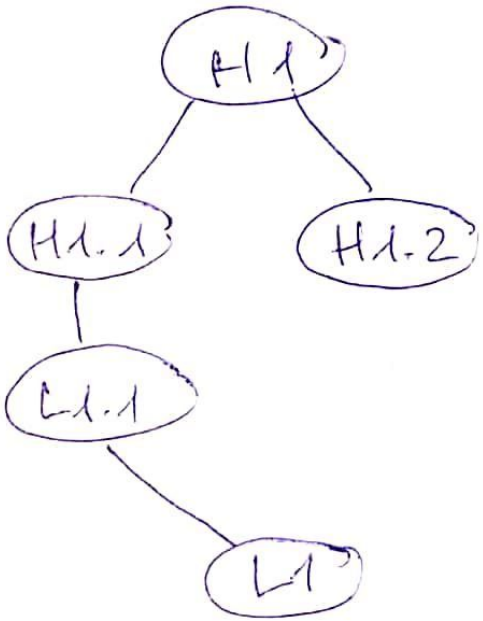
- MAKSYMALNA ODLEGŁOŚĆ MIĘDZY WIERZCHOŁKAMI W T

(zakładam że odległość liczona jest w krawędziach)

```
int max_wysokosc(Node* n, int& akt_max)
{
  if(n == NULL)
    return 0;

  int max_l = max_wysokosc(n->left, akt_max);
  int max_r = max_wysokosc(n->right, akt_max);

  if(max_l + max_r + 1 > akt_max)
    akt_max = max_l + max_r + 1

  if (max_l > max_r)
    return max_l + 1;
  else
    return max_r + 1;
}
```

ZAD. 2

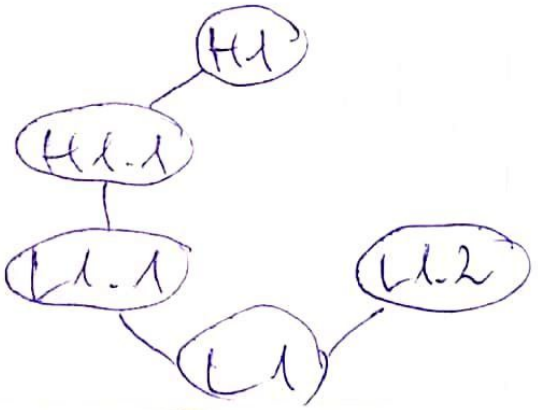




[H1, L1, H1.1, L1.1, H1.2, L1.2]

[H1, L1, H1.1, L1.1, H1.2

Do takiej sytuacji ——————> dojść niemoże



W tablicy na prawym będziemy przechowywać elementy z H i z L.

Dla kopca H:
ojciec: i
syn1: 2i+1
syn2: 2i+3

Dla kopca L:
ojciec: i
syn1: 2i
syn2: 2i+2

Dla dowolnego kopca:

Syn: i

Ojciec: $i/2 \% 2 == i \% 2$ jeśli

TAK → $i/2$

NIE → $i/2-1$

Zatem $i/2-((i/2 \% 2) \wedge (i \% 2))$

Ojciec: i

Syn1: $2i+(i \bmod 2)$

Syn2: $2i+2+(i \bmod 2)$

```
USUŃ-MIN(T[1...n])
    min = T[2]
    T[2] = T[n]
    PRZESUŃ-NIŻEJ(T[1...n], 2)

USUŃ-MAX(T[1,...n])
    max = T[1]
    T[1] = T[n]
    PRZESUŃ-NIŻEJ(T[1,...n], 1)

PRZESUŃ-NIŻEJ(T[1,...n], i)
    k = i
    x = i%2
    if x == 1
        do
            j = k
            parent = j/2%2 == j%2 ? j/2 : j/2-1
            if j>2 && T[parent] < T[k]
                k = parent
            swap(T[j], T[k])
        while(j == k)
    else
        do
            j = k
            parent = j/2%2 == j%2 ? j/2 : j/2-1
            if j>2 && T[parent] > T[k]
                k = parent
            swap(T[j], T[k])
        while j == k
```

```
PRZESUN-W GÓRĘ(T[1,...n], i)
    k = i
    x = i % 2
    if x == 1   // dla kopca H
        do
            j = k
            if 2j+x ≤ n && T[2j+x] > T[k]
                k = 2j + x
            if 2j+2+x ≤ n && T[2j+2+x] > T[k]
                k = 2j+2+x
            swap(T[j], T[k])
        while(j == k)

    if 2k+x > n          // przywracanie do kopca L
        if k < n && T[k] < T[k+1]
            swap(T[k], T[k+1])
            PRZESUN-W GÓRĘ(T[1...n], k+1)
        else
            i = k+1
            nastepny = i/2 % 2 == i % 2 ? i/2 : i/2-1
            if T[k] < T[nastepny]
                swap(T[k], T[nastepny])
                PRZESUN-W GÓRĘ(T[1...n], nastepny)
    else // dla kopca L
        do
            j = k
            if 2·j + x ≤ n && T[2j+x] < T[k]
                k = 2j + x
            if 2j+2+x ≤ n && T[2j+2+x] < T[k]
                k = 2j+2+x
            swap(T[j], T[k])
        while (j == k)

        if 2k+x > n      // przywracanie do kopca H
            if k > 1 && T[k] > T[k-1] && n % 2 == 0
                swap(T[k], T[k-1])
                PRZESUN-W GÓRĘ(T[1,...n], k-1)
        else if n % 2 != 0
            if T[k] > T[2k-1]
                swap(T[k], T[2k-1])
                PRZESUN-NIŻEJ(T[1...n], 2k-1)
```

ZAD. 3

```
wynik = []

for wierzchołek(v) in zbiór wierzchołków:
    stopień[v] = 0;
    sąsiedzi[v] = [];

for krawędź (u,v) in zbiór krawędzi:
    stopień[v]++;
    sąsiedzi[u].append(v);

for wierzchołek(v) in zbiór wierzchołków:
    if (stopień[v] == 0)
        kolejkaPriorytetowa.insert(v)

while(!kolejkaPriorytetowa.empty()):
    min = kolejkaPriorytetowa.findMin();
    wynik.append(min)
    kolejkaPriorytetowa.deleteMin();
    for sąsiad in sąsiedzi[min]:
        stopień[sąsiad]--;
        if (stopień[sąsiad] == 0)
            kolejkaPriorytetowa.insert
                (sąsiad);
```

wagi są rzeczywiste dodatnie

ZAD. 4 NIECH u i v BĘDĄ DWOMA WIERZCHOŁKAMI W GRAFIE NIESKIEROWANYM $G=(V,E;c)$, GDZIE $c: E \to R_+$ JEST FUNKCJĄ WAGOWĄ. MÓWIMY, ŻE DROGA Z $u = u_1, u_2, \ldots, u_{k-1}, u_k = v$ Z u DO v JEST SENSOWNA, JEŚLI DLA KAŻDEGO $i = 2, \ldots, k$ ISTNIEJE DROGA Z u_i DO v KRÓTSZA OD KAŻDEJ DROGI Z u_{i-1} DO v (PRZEZ DŁUGOŚĆ DROGI ROZUMIEMY SUMĘ WAG JEJ KRAWĘDZI).

UTÓŻ ALGORYTM, KTÓRY DLA DANEGO G ORAZ WIERZCHOŁKÓW u i v WYZNACZY LICZBĘ SENSOWNYCH DRÓG Z u DO v.

Aby powiedzieć, że dana droga jest sensowna lub też nie, musimy znać koszty odległości każdego wierzchołka na niej znajdującej się do wierzchołka v.

Korzystając z algorytmu Dijkstry wyznaczymy najmniejszy koszt dojścia do v dla każdego wierzchołka w grafie.

Następnie zaczniemy od wierzchołka początkowego u i przejdziemy w głąb grafu, żeby policzyć ile sensownych dróg z u do v jest. Rekurencyjnie wyznaczamy dla wierzchołków ilość sensownych ścieżek mnożąc wynik przez ilość sensownych ścieżek w poprzednim wierzchołku na ścieżce

```
SCIEZKI(G,v)
    dla i od 0 do liczby wierzchołków - 1
        odwiedzony[i] = false
        sciezki[i] = 0

    sciezki[v] = 1  ->  jedyna sensowna droga
    odwiedzony[v] = true
    odległości[0.... n-1] = DIJKSTRA(G, i, v)

    wyznacz(v)
    return sciezki

WYZNACZ(w)
    dla każdego sąsiada wierzchołka w
        jeżeli odległości[u] < odległości[w]
            jeżeli odwiedzony[u] == false
                wyznacz(u)

    sciezki[w] = sciezki[w] +
                 + sciezki[u]
    odwiedzony[w] = true
```

ZAD. 5

I) IDEA: Sortujemy tablicę wierzchołków topologicznie, następnie dla każdego wierzchołka przypisujemy długość drogi taką, jaką ma, lub długość drogi jego "ojca" + 1.

Gdy tablica jest posortowana topologicznie to mamy już ogarnięte przodków wierzchołka, którego aktualnie sprawdzamy:

```
VisitNode(u):
    Oznacz u jako odwiedzony
    dla każdego wierzchołka v na liście
    sąsiedztwa u:
        jeżeli v nieodwiedzony:
            VisitNode(v)
    na początek listy L wstaw u

TopologicalSort(Graf G):
    L ← lista posortowanych topologicznie
    wierzchołków
    dla każdego wierzchołka u z grafu G:
        oznacz u jako nieodwiedzony
    dla każdego wierzchołka u z grafu G:
        jeżeli u nieodwiedzony:
            VisitedNode(u)
```

```
maxLength (Graf G):
    sortedArray[] ← TopologicalSort(G)
    D[] ← 0
    dla kazdego wierzchołka v w sortedArray[]:
        dla kazdego wierzchołka u, ktory
        jest sąsiadem v
            D[u] ← max(D[u]; D[v]+1);
    return max(D[]);
```

II) IDEA: Zamiast trzymania dla wierzchołka v tylko wartości najdłuższej drogi trzymamy parę <wartość, rodzic> gdzie rodzic to wierzchołek, z którego przeszliśmy do v zwiększając maks[v].

```
printPath(v)
    if (v.second == null):
        print(v)
    else
        print(printPath(v.second) + " " + v)

maxLength(Graf G)
    sortedArray[] ← TopologicalSort(G)
    P[] ← (0,0)
    dla kazdego wierzchołka v w sortedArray[]:
        dla kazdego wierzchołka u, ktory jest
        sąsiadem v
            if (P[u].first < P[v].first+1):
                P[u].second ← P[v]
                P[u].first ← P[v].first+1
    maxPath ← max(P[].first);
    printPath(maxPath)
```

zł. czasowa $O(|V|+E)$
zł. pam. $O(|V|+E)$

LISTA 1 – ĆWICZENIA

ZAD 6. DANY JEST NIEMALEJĄCY CIĄG n LICZB CAŁKOWITYCH DODATNICH $a_1 \leq a_2 \leq \ldots \leq a_n$. WOLNO NAM MODYFIKOWAĆ TEN CIĄG ZA POMOCĄ NASTĘPUJĄCEJ OPERACJI: WYBIERAMY DWA ELEMENTY a_i, a_j SPEŁNIAJĄCE $2a_i \leq a_j$ I WYKREŚLAMY JE OBA Z CIĄGU. UTÓŻ ALGORYTM OBLICZAJĄCY, ILE CO NAJWYŻEJ ELEMENTÓW MOŻEMY W TEN SPOSÓB USUNĄĆ

IDEA:

Dzielimy n elementową tablicę na dwie tablice L i P. Element ciągu oznaczamy jako a_i. Początek tablicy L to indeks 0, a początek tablicy P to indeks $\lceil n/2 \rceil$.

Lewy wskaźnik, którym będziemy chodzić po tablicy L będziemy zwiększać o 1 tylko wtedy gdy dany element w tablicy P jest conajmniej dwukrotnie większy od obecnego elementu w tablicy L (wskaźnik prawy zwiększamy o jeden zawsze); w przypadku gdy inkrementujemy wskaźnik to counter zwiększamy dwa. Counter to zmienna, która przechowuje ile maks elementów można usunąć.

```
FUNKCJA (T[], n){
    counter = 0;
    left = 0;
    right = n/2

    while (left < n/2 && right < n)
        if(2 * T[left] ≤ T[right]){
            counter += 2
            left++;
        }
        right++;
}
return counter;
}
```