ZESTAW I

1. Zadanie polega na zamianie nieuporządkowanej listy jednokierunkowej z powtórzeniami w postać skompresowaną zawierającą te same dane. Np. jeśli lista `L` ma postać

$$4 \to 3 \to 5 \to 3 \to 4 \to 1 \to 2 \to 5 \to 4 \to 2 \to 5 \to 2 \to 3 \to 3\,,$$

wówczas postać skompresowana będzie następująca `(x,n)`:

$$(2,3) \to (1,1) \to (5,3) \to (3,4) \to (4,3)\,.$$

gdzie `n` oznacza liczbę powtórzeń wartości `x` w oryginalnej liście `L`. Elementy listy skompresowanej są typu `elk`:

```
struct ek { int x;  int n;  struct ek *nast } elk;
```

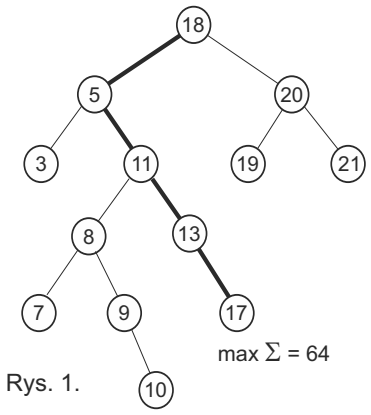
Rozwiązanie powinno mieć postać funkcji `void kompresuj(wel *L, welk *Q)`, gdzie `welk` jest typem wskaźnika na elementy listy skompresowanej. Funkcja ta tworzy listę `*Q` na podstawie `*L`. Można przyjąć, że lista `*L` ulega zwolnieniu, to znaczy, że kolejne elementy `*L` są z niej usuwane podczas tworzenia odpowiadających im elementów `Q`.

2. Napisz funkcję `void invert(wel C)`, która odwraca kierunek wskaźników w *cyklicznej* liście jednokierunkowej `C` bez tworzenia nowych elementów. Wskaźnik `C` na głowę listy nie zmienia swojej wartości. Np. jeśli lista zawiera kolejno liczby 1, 2, 3, 4, odwrócona lista też zaczyna się od 1, ale wskaźniki na następne elementy pokazują kolejno 4, 3, 2. Ostatni element 2 wskazuje cyklicznie na głowę 1.

3. Napisz funkcję `wel wspolne(wel P, wel Q)`, której argumentami są wskaźniki na 2 jednokierunkowe listy nieuporządkowane bez powtarzających się elementów, a zwracaną wartością jest wskaźnik na nową listę, która zawiera wszystkie wspólne elementy list `P` i `Q`. Jeśli np. `P` zawiera kolejno elementy 4,2,7,1,5,3, a `Q` zawiera liczby 1,8,5,9,2,10,4, wtedy lista `R` utworzona jako `R = wspolne(P,Q)` zawierać będzie liczby 5,1,2,4. Listy `P` i `Q` pozostają niezmienione.

4. Napisz funkcję logiczną `int BST(weld T)`, która sprawdza czy jej argument jest drzewem uporządkowanym BST. Rozwiązanie może być rekurencyjne lub iteracyjne. Można posłużyć się funkcjami `mindrzewa(T)` i `maxdrzewa(T)` bez konieczności pisania ich kodów. Zakładamy, że zwracają one wartości tych węzłów `T`, w których, zgodnie ze strukturą BST, powinny znajdować się odpowiednio największe i najmniejsze elementy `T`. Rozwiązanie iteracyjne może także posłużyć się wywołaniem funkcji `nastepnik(p)`, która zwraca *wskaźnik* na węzeł w `T`, gdzie w strukturze BST znajduje się następnik elementu wskazywanego przez `p`.

5. Napisz kod funkcji `int max_suma(weld T)`, która oblicza największą z sum elementów drzewa BST liczonych wzdłuż ścieżek z korzenia do liści.



Rys. 1.

ZESTAW II

1. Idea sortowania przez scalanie (merge sort) polega na rekurencyjnym podziale tablicy na coraz to mniejsze segmenty, oddzielne ich sortowanie, a następnie systematyczne scalanie par posortowanych segmentów w jeden większy. Efektywność algorytmu bierze się z faktu, że scalenie dwóch posortowanych oddzielnie ciągów wymaga liniowej względem sumy ich długości liczby kroków.

Napisz funkcję `void scalaj(int T[], int l, int c, int p)`, która dokonuje połączenia dwóch posortowanych rosnąco segmentów tablicy `T`: od `T[l]` do `T[c]` oraz od `T[c+1]` do `T[p]`. Można użyć pomocniczych lokalnych tablic odpowiedniej wielkości. Wynik — jeden posortowany rosnąco ciąg zapisywany jest na powrót w tablicy `T`, od `T[l]` do `T[p]`.

2. Czy zapamiętanie i wykorzystanie miejsca ostatnio wykonanej zamiany w przebiegu sortowania bąbelkowego wpływa na łączną liczbę: a) tylko podstawień, b) tylko porównań, c) na obydwie liczby, d) nie wpływa na żadną z nich?

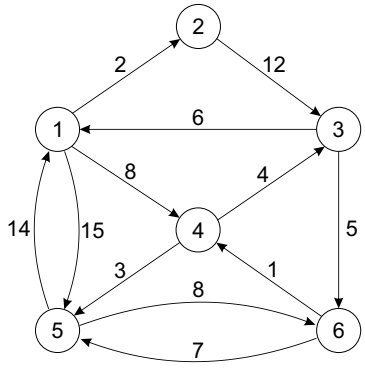
3. Wykorzystaj schemat BFS przeszukiwania grafu do napisania funkcji, która wyznacza najkrótsze ścieżki w grafie z wierzchołka startowego `s` do wszystkich pozostałych wierzchołków. W tym wypadku długość ścieżki oznacza po prostu liczbę zawartych w niej krawędzi. Graf zadany jest w postaci list sąsiedztwa. Strukturę najkrótszych ścieżek, podobnie jak w algorytmie Dijkstry, można zapisać i później odtworzyć z tablicy `int Poprz[n]`, gdzie `n` jest liczbą wierzchołków w grafie oraz `Poprz[u] = v` jeśli `v` jest poprzednikiem `u` na najkrótszej ścieżce z `s`. Dla wierzchołka startowego mamy `Poprz[s]=0`.

Przypomnijmy, że BFS pobiera kolejne wierzchołki `u` z kolejki `Q`, dopisując na jej koniec nieodwiedzonych jeszcze sąsiadów `u`. Na początku `Q` zawiera wierzchołek startowy `s`, natomiast algorytm kończy pracę, gdy `Q` stanie się pusta. Jeśli pozostają wciąż nieodwiedzone wierzchołki, nie są one osiągalne z wierzchołka `s`, a więc graf nie jest spójny.

4. Wierzchołki grafu ponumerowane są liczbami `u = 1, 2, ... n`, a jego struktura opisana jest przez listy sąsiedztwa, `S[u]` = lista sąsiadów wierzchołka `u`. W tablicy `Odwiedzony[u]` rejestrujemy fakt odwiedzenia wierzchołka `u` w procesie DFS. Napisz kod zmodyfikowanej procedury DFS tak, aby wykrywała ona istnienie cykli w grafie, to jest powinna zwracać wartość 1, gdy graf zawiera cykle lub 0 w przeciwnym razie. Istnienie cyklu wykrywane jest wtedy, gdy proces DFS napotyka wcześniej odwiedzony wierzchołek `v`, który nie jest bezpośrednim rodzicem aktualnie przetwarzanego wierzchołka `u` w drzewie DFS.

5. Przeprowadź symulację działania algorytmu Dijkstry na skierowanym grafie `G` o strukturze przedstawionej na rysunku. Wierzchołkiem startowym jest najpierw a) `s=1`, a potem b) `s=5`. W każdym przypadku podaj kolejność, w jakiej wierzchołki kwalifikowane będą jako odwiedzone. Wypisz przebieg najkrótszych ścieżek z `s` do pozostałych wierzchołków. Struktura list sąsiedztwa jest następująca:

S[1] → 2, 5, 4	S[2] → 3	S[3] → 1, 6
S[4] → 3, 5	S[5] → 6, 1	S[6] → 5, 4



Rozwiązania

I-1. Pomocnicze funkcje: usuwanie elementów z `*L` i dodawanie nowych do `*Q`:

```
int usun(wel *P){
    int y = (*P)->x;
    wel S = *P;
    *P = (*P)->nast;
    free(S);
    return y;
}
```

```
void dodaj(welk *Q, int y, int m){
    welk R = malloc(sizeof(elk));
    R->x = y;
    R->n = m;
    R->nast = *Q;
    *Q = R;
}
```

```
void kompresuj(wel *L, welk *Q}{
   wel *K;
   int y, m;
   *Q = NULL;
   while (*L != NULL){
       K = L;
       m = 1;
       y = usun(K);
       while (*K != NULL)
           if (*K->x == y) { usun(K); m++; }
           else { K = &((*K)->nast); }
       dodaj(Q, y, m);
   }
}
```

I-2.
```
void invert(wel C}{
   wel S=C, S1, S2;
   if ((C!=NULL) && (C->nast!=C)){
      do {
         S1 = S->nast;
         S2 = S1->nast;
         S1->nast = S;
         S = S1;
      }  while (S!=C);
   }
}
```

I-3.
```
wel wspolne(wel P, wel Q){
   wel R, S, T=NULL;
   while (P!=NULL){
       R = Q;
       while ((R!=NULL) && (P->x!=R->x)) R = R->nast;
       if (R!=NULL) {
          S = malloc(sizeof(el));
          S->x =  R->x;
          S->nast = T;
          T = S;
       }
       P = P->nast;
   }
   return T;
}
```

I-4.
```
int BST(weld T}{                 //  rozwiązanie rekurencyjne
   int k=1;
   if (T != NULL) {
      if (T->l != NULL) k = BST(T->l) && (T->x >= maxdrzewa(T->l));
      if (T->p != NULL) k = k && BST(T->p) && (T->x <= mindrzewa(T->p));
   }
   return k;
}

int BST (weld T){                //  rozwiązanie iteracyjne
    weld Q=T, S;
    if (T!=NULL){
        if (T->l != NULL){
            Q = T->l;                        //  ustawiamy Q na węźle, który
            while (Q->p != NULL) Q = Q->p;   //  w BST zawiera minimum drzewa
        }
        S = nastepnik(Q);
        while (S!=NULL) {
            if (Q->x > S->x) return 0;
            Q = S;
            S =  nastepnik(Q);
        }
    }
    return 1;
}
```

I-5.
```
int max_suma(weld T}{
   int j,k;
   if (T==NULL) return 0;
   j = max_suma(T->l);
   k = max_suma(T->p);
   return T->x + (j>k) ? j : k;
}
```

II-1.
```
void scalaj(int T[], int l, int c, int p}{
   int i=l, j=c+1, k=0;
   int S[p-l+1];
   while ((i<=c) && (j<=p))
      if (T[i] <= T[j]) { S[k++] = T[i++]; }
      else { S[k++] = T[j++]; }
   while (i<=c) S[k++] = T[i++];
   while (j<=p) S[k++] = T[j++];
   for (i=0; i<k; i++) T[l+i] = S[i];
}
```

II-2. Modyfikacja wpływa tylko na liczbę wykonywanych porownań. Liczba przestawień jest taka sama i zależy jedynie od początkowego ułożenia liczb, jeśli porządkowanie odbywa się poprzez zamianę sąsiadów `T[k]` z `T[k+1]`. Zapamiętanie i wykorzystanie miejsca ostatniej zamiany ogranicza liczę porównań, nie wykonując ich niepotrzebnie między elementami ustawionymi już zgodnie z porządkiem w prawej części tablicy.

II-3. Zakładam, że `S[u]` jest wskaźnikiem na listę sąsiadów wierzchołka `u`, a `Q` jest kolejką przechowującą numery wierzchołków ze standardowymi operacjami `pop(Q)`, `push(Q,v)` i `empty(Q)`.

```
void sciezki_BFS(wel S[], int Odw[], int Poprz[], int n, int s}{
    int u,v;
    for (v=0; v<n; v++) Poprz[v] = Odw[v] = 0;
    push(Q, s);
    Odw[s] = 1;
    while (!empty(Q)) {
       u = pop(Q);
       for (v = S[u]; v!=NULL; v=v->nast)
          if (!Odw[v]){
              Odw[v] = 1;
              push(Q, v);
              Poprz[v] = u;
          }
       }
    }
    printf("podaj wierzcholek docelowy: ");
    scanf("%i", &u);
    printf("sciezka z %i do %i (odwrocona): ", s, u);
    while (u!=0){
       printf("%i ", u);
       u = Poprz[u];
    }
}
```

II-4.
```
int DFS_cykl(int u, int r){
    int v, c=0;
    Odwiedzony[u] = 1;
    v = S[u];
    while ((v!=NULL) && (c==0)){
        if (Odwiedzony[v]){ if (v!=r) c = 1; }
        else { c = DFS_cykl(v, u); }
        v = v->nast;
    }
    return c;
}
```

II-5. Kolejność kwalifikowania wierzchołków jako odwiedzone:

a) 1, 2, 4, 5, 3, 6 b) 5, 6, 4, 3, 1, 2