AISD FUNKCJE PODSTAWOWE (KOLOKWIUM 1)

1. **Stos:**
   - `void push(elem * &stos, int x)`
   - `int pop(elem * &stos)`
   - `int topEl(elem * &stos)`
   - `bool isEmpty(elem * stos)`
   - `void print(elem* stos)`
2. **Kolejka:**
   - `void add(elem * &pocz, elem * &kon, int x)`
   - `int next(elem * & pocz, elem * &kon)`
   - `int firstEl(elem* pocz)`
   - `bool isEmpty(elem * pocz)`
   - `void print(elem* pocz)`
3. **Lista jednokierunkowa:**
   - `void insert(int x, int i, elem* &lista)`
   - `void remove(int i, elem* &lista)`
   - `int read(int i, elem* lista)`
   - `int size(elem* lista)`
   - `void print(elem* lista)`
   - `elem* search(int x, elem* lista)`
4. **Lista dwukierunkowa:**
   - `void insert(int x, int i, elem* &lista)`
   - `void remove(int i, elem* &lista)`

ALGORYTMY I STRUKTURY DANYCH – KOLOKWIUM I

Struktura:

```
struct elem{
   int dane;
   elem* nast;
 };
```

:: 01 STOS:

1. Dodawanie elementu na stos:

```
void push(elem* &stos, int x)
  { elem* nowy = new elem;
    nowy->dane = x;
    nowy->nast = stos;
    stos = nowy; }
```

2. Usuwanie elementu ze stosu:

```
int pop(elem* &stos)
  { if (stos!=NULL)
       {elem* stary = stos;
        int wynik = stos->dane;
        stos = stos->nast;
        delete stary;
        return wynik;
        }
    else throw runtime_error(„Pusty stos”); }
```

3. Wyświetlanie ostatniego elementu na stosie:

```
int topEl(elem* &stos)
  { if (stos!=NULL)
       {int wynik = stos->dane;
        return wynik; }
    else throw runtime_error(„Pusty stos”);  }
```

4. Sprawdzenie, czy stos jest pusty:

```
bool isEmpty(elem* stos)
  { if (stos==NULL) return true;
      else return false; }
```

5. Wydruk stosu:

```
void print(elem* stos) {
     elem* tmp = stos;
     while (tmp != NULL) {
          if (tmp != stos) {
               cout << " <- ";
          }
          cout << tmp->dane;
          tmp = tmp->nast;
     }
     cout << endl; }
```

:: 02 KOLEJKA:

1. Dodanie elementu do kolejki:

```
void add(elem* &pocz, elem* &kon, int x)
   { if(kon!=NULL)
      {elem* nowy = new elem;
       nowy->dane = x;
       nowy->nast = NULL;
       kon->nast = nowy;
       kon = nowy;}
     else {
       elem* nowy = new elem;
       nowy->dane = x;
       nowy->nast = NULL;
       kon = nowy;
       pocz = nowy;}
   }
```

2. Usuwanie elementu z kolejki:

```
int next(elem* & pocz, elem* &kon)
   { if(pocz!=NULL)
       {elem* stary = pocz;
        int wynik = stary->dane;
        if (pocz==kon) kon=NULL;
        pocz = stary->nast;
        delete stary;
        return wynik;
       }
    else { throw runtime_error(„Pusta kolejka”); }
  }
```

3. Wyświetlenie pierwszego elementu kolejki:

```
int firstEl(elem* pocz)
  { if(pocz!=NULL)
      {int wynik = pocz->dane;
       return wynik;
      }
    else throw runtime_error(„Pusta kolejka”);
  }
```

4. Sprawdzenie, czy kolejka jest pusta:

```
bool isEmpty(elem* pocz)
   { if (pocz==NULL) return true;
     else return false;
   }
```

5. Wyświetlenie kolejki:

```
void print(elem* pocz) {
     elem* tmp = pocz;
     while (tmp != NULL) {
          if (tmp != pocz) {
               cout << " <- ";
          }
          cout << tmp->dane;
          tmp = tmp->nast;
          }
     cout << endl; }
```

:: 03 KOLEJKA I STOS:

1. Kolejka za pomocą dwóch stosów:

```
void add(elem* &stos1, elem* &stos2, int x)
   { push(stos2,x); }
```

```
int next(elem* &stos1, elem* &stos2)
  { if(stos1!=NULL) return pop(stos1);
      else {
          while (stos2!=NULL)
            {push(stos1,pop(stos2); }
            if(stos1!=NULL) return pop(stos1);
                else throw runtime_error(„Pusta kolejka”);
               } }
```

2. Kolejka przy użyciu tablicy:

```
const int N = …; //rozmiar tablicy
int kolejka[N];
int kon = 0, pocz = 0, rozm = 0;
```

```
void add(int x)
  { if (rozm==N) cout<<”Kolejka jest pełna”;
      else {
                kolejka[kon] = x;
                rozm++;
                kon++;
                if (kon==N) kon=0;
             } }
```

```
int next()
  { if (rozm==0) throw runtime_error(„Pusty stos”);
      else { int wynik = kolejka[pocz];
                rozm--;
                pocz++;
                if (pocz==N) pocz=0;
                return wynik; }}
```

3. Stos przy użyciu tablicy:

```
const int N = …; //rozmiar tablicy
int stos [N];
int rozm = 0;
```

```
void push(int x)
  { if (rozm==N) cout<<”Stos jest pełny”;
      else {
                stos [rozm] = x;
                rozm++;
             } }
```

```
int pop()
  { if (rozm==0) throw runtime_error(„Pusta stos”);
      else {rozm--;
                int wynik = stos[rozm];
                return wynik; }}
```

4. Odwracanie porządku elementów na stosie przy użyciu kolejki:

```
void obroc(elem* &stos, elem* &pocz, elem* &kon)
  {while(stos!=NULL)
      {add(pocz, kon, pop(stos));
      }
    while(kon!=NULL)
```

```
      {push(stos,next(pocz,kon));
     }
  }
```

5. Odwracanie porządku elementów na stosie przy użyciu drugiego stosu:

```
void obroc(elem* &stos1, elem* &stos2)
   {int tmp = 0;
    elem* stop = NULL;

   while(stop!=stos1)
      {tmp=pop(stos1);
       while(stos1!=stop)
            {push(stos2,pop(stos1));}
       push(stos1,tmp);
       stop=stos1;
       while(stos2!=NULL)
            {push(stos1,pop(stos2));}
     }}
```

6. Uporządkowanie elementów na stosie według malejących wartości, korzystając z jednego dodatkowego stosu i kilku zmiennych lokalnych.

```
  void minimalna_kolejnosc(elem* &stos1,elem* &stos2)
  {elem* stop = NULL;
   int min = 0;
   while(stos1!=stop)
     {min = pop(stos1);
      while(stos1!=stop)
        {if (topEl(stos1)<min)
             {push(stos2,min);
              min=pop(stos2);
             }
         else push(stos2,pop(stos1));
        }
       push(stos1,min);
       stop=stos1;
       while(stos2!=NULL)
          push(stos1,pop(stos2));
     }
  }
```

7. Przeniesienie elementów ze stosu S1 na stos S2 tak, aby był zachowany porządek. Mamy do dyspozycji 1 dodatkowy stos.

```
  void zmiana_stosow(elem* &stos1,elem* &stos2,elem* &stos3)
  { //stos3 - stos pomocniczy
   while(stos1!=NULL)
     push(stos3,pop(stos1));
   while(stos3!=NULL)
     push(stos2,pop(stos3));
  }
```

8. Przeniesienie elementów ze stosu S1 na stos S2 tak, aby był zachowany porządek. Mamy do dyspozycji tylko zmienne lokalne.

```
  void move(elem* &stos1, elem* &stos2) {
     int x, i = 0;
     while (stos1 != NULL) {
          while (stos1 != NULL) {
```

```
                push(stos2, pop(stos1));
                i++;
            }
            int x = pop(stos2);
            i--;
            while (i > 0) {
                push(stos1, pop(stos2));
                i--;
            }
            push(stos2, x);
        }
    }
```

:: 04 LISTY JEDNOKIERUNKOWE:

1. Dodawanie elementu na i-tą pozycję listy:

```
void insert(int x, int i, elem* &lista)
   {if (i==1)
      {elem* nowy = new elem;
       nowy->dane = x;
       nowy->nast = lista;
       lista = nowy;}
   else if (i>1)
     { elem* nowy = new elem;
      nowy->dane = x;

      elem* poprz = lista;
      int licznik = 0;
      while(licznik!=(i-2))
           {licznik++;
            poprz=poprz->nast;
            if (poprz==NULL) throw runtime_error(„Za krotka
lista!);}

     nowy->nast = poprz->nast;
     poprz->nast = nowy;
   }   else cout<<”Błędne dane”;
}
```

2. Usuwanie elementu z i-tej pozycji listy:

```
void remove(int i, elem* &lista)
   {if (lista!=NULL)
      {if (i==1)
        {elem* stary = lista;
         lista = lista->nast;
         delete stary;}
     else if (i>1)
         { elem* poprz = lista;
           int licznik = 0;
           while(licznik!=(i-2))
                 {licznik++;
                  poprz=poprz->nast;
                  if (poprz==NULL) throw runtime_error(„Za krotka
lista!);}
          elem* stary = poprz->nast;
```

```
          if (stary==NULL) throw runtime_error(„Brak elementu do
usuniecia”);
          poprz->nast = stary->nast;
          delete stary;
         } else cout<<”Błędne dane”;
 } else cout<<”Pusta lista!” }
```

3. Odczyt i-tej pozycji listy:

```
int read(int i, elem* lista)
   {if (lista!=NULL)
      {if (i>=1)
         { elem* wyn = lista;
           int licznik = 0;
           while(licznik!=(i-1))
                 {licznik++;
                  wyn=wyn->nast;
                  if (wyn==NULL) throw runtime_error(„Za krotka
lista!);}
          int wynik = wyn->dane;
      return wynik;}
    else cout<<”Błędne dane”
  } else cout<<”Pusta lista!” }
```

4. Obliczanie wielkości listy:

```
int size(elem* lista)
  { if (lista==NULL) return 0;
    else {
        int licznik=0;
        while(lista!=NULL)
             {licznik++;
              lista=lista->nast;}
        return licznik; } }
```

5. Wypisanie listy:

```
void print(elem* lista)
  {if(lista==NULL) cout<<”Pusta lista”;
   else{
       while(lista!=NULL)
        {cout<<lista->dane<<endl;
          lista=lista->nast;}
   }
```

6. Wypisanie elementów listy w odwróconej kolejności, korzystając ze stosu:

```
void printReversedWithStack(elem* lista,elem* &stos) {
     for (int i = 0; i < size(lista); ++i) {
          push(stos,read(i, a));
     }
     while (stos!=NULL) {
          cout << "Element z stosu: " << pop(stos); << endl;
     }
}
```

7. Wypisanie elementów listy w odwróconej kolejności, bez struktury danych:

```
void printReversed(elem* lista)
     {for (int i = size(lista) ; i != 0; i--) {
          cout << "Element: " << read(i, lista) << endl;
     }
}
```

```
void printReversedRec(elem* lista)
             {if(lista!=NULL)
                 { printReversedRec(lista->nast);
                   cout<<lista->dane<<endl;
                  }
               }
```

8. Usuwanie wszystkich elementów listy:

```
void destroy(elem* &lista)
   {if (lista==NULL) cout<<"Pusta lista";
    else {elem* temp = lista;
              while(temp!=NULL)
                 {elem* stary = temp;
                  temp = temp->nast;
                  delete stary;
                 }
            } }
```

```
void destroy_rec(elem* lista)
   {if(lista!=NULL)
       {destroy_rec(lista->nast);
        delete lista;
        lista=NULL;
       }
   }
```

9. Znalezienie wskaźnika do elementu, w którym znajduje się liczba x:

```
elem* search(int x, elem* lista)
  {elem* wynik = lista;
    while(lista!=NULL)
       {if(wynik->dane==x) return wynik;
        wynik=wynik->nast;}
     throw runtime_error(„Nie znaleziono");
   }
```

10. Suma dwóch wielomianów:

```
struct elem{
   int wykl;
   int wspolcz;
   elem* nast;}
```

```
void insert(int wykl, int wspolcz, int i, elem* &lista)
```

```
elem* polyadd(elem* w1, elem* w2) {
elem* t = NULL;
int i = 0;
while (w1 != NULL || w2 != NULL) {
      if (w1 != NULL && w2 != NULL && w1->wykl == w2->wykl) {
            insert(w1->wykl, w1->wspolcz + w2->wspolcz, i++, t);
            w1 = w1->nast;
            w2 = w2->nast;
      }
      else if ((w1 != NULL && w2 != NULL && w1->wykl > w2->wykl) || (w1
!= NULL && w2 == NULL)) {
            insert(w1->wykl, w1->wspolcz, i++, t);
            w1 = w1->nast;
      }
      else if ((w1 != NULL && w2 != NULL && w1->wykl < w2->wykl) || (w1
== NULL && w2 != NULL)) {
            insert(w2->wykl, w2->wspolcz, i++, t);
```

```
            w2 = w2->nast;
      }
}
return t;
}
```

11. Ułamki Farey'a:

```
struct elem_farey{
    int licznik;
    int mianownik;
    elem_farey* nast;}
```

```
elem_farey* ulamek_fareya(int n)
   {if (n==1)
      {elem_farey* pierwszy = new elem_farey;
       elem_farey* drugi = new elem_farey;
       pierwszy->licznik = 0;
       pierwszy->mianownik=1;
       drugi->licznik=1;
       drugi->mianownik=1;
       pierwszy->nast=drugi;
       drugi->nast=NULL;
 } else {elem_farey* temp = ulamek_farey(n-1);
  elem_farey* kopia = temp;
  elem_farey* nastepny = temp->nast;
  while(nastepny!=NULL)
     {if((temp->mianownik+nastepny->mianownik)<=n)
         {elem_farey* nowy = new elem_farey;
          nowy->licznik=kopia->licznik+nastepny->licznik;
          nowy->mianownik= kopia->mianownik+nastepny->mianownik;
          nowy->nast=nastepny;
          temp->nast=nowy;
         } temp=nastepny;
           nastepny=temp->nast;
     return kopia;} }
```

Struktura:

```
struct elem{
   int dane;
   elem* nast;
   elem* poprz;};
```

:: 04 LISTY DWUKIERUNKOWE I CYKLICZNE:

1. Wstawienie elementu x do listy dwukierunkowej:

```
void insert(int x, int i, elem* &lista)
   {if (i==1)
      {elem* nowy = new elem;
       nowy->dane = x;
       nowy->nast = lista;
       nowy->poprz=NULL;
      if(lista!=NULL) lista->poprz=nowy;
      lista=nowy;
      } else if (i>1)
       {  elem* temp = lista;
          for(int k = 1 ; k < i ; k++)
```

```
            {temp = temp->nast;}
          elem* nowy = new elem;
          nowy->dane = x;
          elem* nastepny = temp->nast;
          temp->nast = nowy;
          if(nastepny!=NULL)
            nastepny->poprz = nowy;
            nowy->poprz = temp;
            nowy->nast = nastepny;
            nastepny = nowy;}
```

2. Usuwanie elementu z listy dwukierunkowej:

```
void remove(int i, elem* &lista)
  {if (i==1)
      {elem* wsk = lista;
       lista = lista->nast;
       if (lista!=NULL)
            lista->poprz = NULL;
       delete wsk;}
   else {
       elem* wsk = lista;
       for( int j=1;j<=i;j++)
           wsk=wsk->nast;
      wsk->poprz->nast = wsk->nast;
      if(wsk->nast!=NULL)
         wsk->nast->poprz = wsk->poprz;
     delete wsk;
  }
```

3. Odwracanie listy dwukierunkowej:

```
void reverse(elem* &lista)
  {while(lista!=NULL)
      {elem* kopia = lista->nast;
       lista->nast = lista->porz;
       lista->poprz = kopia;
       lista = kopia;
       lista=lista->nast;}
  }
```

4. Procedura przekształcająca listę jednokierunkową w listę cykliczną:

```
void to_cycle(elem* lista)
  {elem* pierwszy = lista;
   while(lista->nast!=NULL)
       lista=lista->nast;
  lista->nast = pierwszy;}
```

5. Zmiana kierunku wskaźników listy cyklicznej jednokierunkowej:

```
Reverse_cyclic(elem* &lista)
 {elem*start=lista->nast.;
   elem*kopiaP=lista;
   elem*kopiaN=NULL;
   lista=lista->nast;
    do  { kopiaN=lista->nast;   lista->nast=kopiaP;
kopiaP=lista;   Lista=kopiaN; }
            while(lista!=start);
```

6. Notacja beznawiasowa:

```
struct elem {
      char dane;
      elem* nast; };
```

```
void insert(char x, int i, elem* &a) {
      if (i < 0) {
            string s = "Indeks z poza zakresu.";
            throw s; }
      if (((int) x < 97 || (int) x > 122) && (int) x != 42 && (int) x
43 && (int) x != 45 && (int) x != 47) {
            string s = "Niepoprawny symbol.";
            throw s;}
      elem* e = new elem;
      e->dane = x;
      e->nast = NULL;
      if (i == 0) {
            e->nast = a;
            a = e;}
      else {
            elem* t = a;
            for (i -= 1; i > 0; --i) {
                  t = t->nast;
                  if (t == NULL) {
                        string s = "Indeks z poza zakresu.";
                        throw s;} }
            e->nast = t->nast;
            t->nast = e; } }
```

```
int size(elem* a) {
      if (a == NULL)  return 0;
      elem* tmp = a;
      int i = 0;
      while (tmp != NULL) {
            tmp = tmp->nast;
            i++;}
      return i; }
```

```
bool sprawdz(elem* wsk) {
      int i = 1;
      while (wsk != NULL) {
            i--;
            if (i < 0)  return false;
            if (wsk->dane == '+' || wsk->dane == '-' || wsk->dane ==
'*' || wsk->dane == '/')
                  i += 2;
            wsk = wsk->nast;
                        }
      return i == 0;
           }
```