AiSD Laboratorium 3

Wskazówki do zadań

Wskazówka do zadania 1.

Przyjmujemy, że T=short.

Węzeł listy jednokierunkowej SLLNode zawiera pola:

- SLLNode* next wskaźnik na następny węzeł
- T data rzeczywiste dane

Węzeł listy dwukierunkowej DLLNode zawiera pola:

- DLL_Node* next wskaźnik na następny węzeł
- DLL_Node* prev wskaźnik na poprzedni węzeł
- T data rzeczywiste dane

Deklaracja jako `struct`

```
// SLLNode_struct.h
struct SLLNode{

    SLLNode* next;
    T data;
};

// DLLNode_struct.h
struct DLLNode{

    DLLNode* next;
    DLLNode* prev;
    T data;
};
```

Uwaga W C++ nie ma różnicy pomiędzy słowami kluczowymi `struct` oraz `class` z wyjątkiem tego, że w pierwszym przypadku wszystkie składowe są domyślnie publiczne a w drugim prywatne. Zatem `struct` deklaruje klasę taką samą jak przy użyciu `class`. Z punktu widzenia semantyki kodu deklarowanie czegoś jako `struct` może wyrazić intencję programisty, aby był to zbiór danych bez operacji (lub z minimalną ich ilością). Deklarowanie jako `class` wyraża intencję, że dany obiekt poza polami bedzie zawierał róznież metody.

Implementacja jako `class`

```
// SSL_Node.h
class SLLNode{

public:
    using T = short;

    SLLNode* next;
    T data;

    SLLNode() = default;                    // konstruktor domyślny
    SLLNode(SLLNode* n,const T& data); // jawny konstruktor
    ~SLLNode() = default;                   // destruktor domyślny

};

// SSL_Node.cpp
```

```
    SLLNode::SLLNode(SLLNode* n,const T& d):
            data(d), next(n)      // lista inicjalizatorów konstruktora
             { };

/*
równoważny sposób w tym przypadku (w ogólności nie)
    SLLNode::SLLNode(SLLNode* n,const T& d){
        next = n;
        data = d;
    };

*/
};
```

```
//DLLNode.h
class DLLNode
{
public:

using T = short;

    DLLNode() = default;
    DLLNode(DLLNode* n, DLLNode* p,const T& el);

    ~DLLNode() = default;

    DLLNode* prev;
    DLLNode* next;
    T data;
};

//DLLNode.cpp

DLLNode::DLLNode(DLLNode* n, DLLNode* p, const T& d ): next(n), prev(p), data(d) { }
```

Patrz folder zadanie_1.

Wskazówka do zadania 2.

Bezpośrednie rozwiązanie tego zadania polega na ręcznym przypisaniu wartości do każdego węzła. Wyświetlanie również można wykonac ręcznie.

Jednak do wyswietlania zgodnie z połączeniem wygodniej jest użyć pętli `while`,

```
// deklaracja
SLLNode n1;
SLLNode n2;
SLLNode n3;
...
// inicjalizacja
 n1.next = &n3; n1.data=1;
 n3.next = &n6; n1.data=3;
 n6.next = &n9; n1.data=6;
...
// wyswietlanie zgodnie z numeracją
    std::cout << "Zgodnie z numeracją"
              << n1.data << " "
              << n2.data << " "
...
// wyswietlanie zgodnie z połączniem

AiSD::SLLNode* current = arr; // lub  AiSD::SLLNode* current = &arr[0];

    while(current != nullptr){
        std::cout << current->data << " -> "; // " -> " opcjonalnie dla ładności
        current = current->next;
    }
    std::cout << "NIL"; //opcjonalnie dla ładności
```

Dzięki temu, że kolejność węzłów oraz przechowywane wartości dane są wzorem można zaproponować rozwiązanie bardziej elastyczne, oparte na tablicy:

```
const size_t N = 10;
AiSD::SLLNode arr[N];

    for(int i = 0; i<N; i++){
        arr[i].next = &arr[ (i+3) % 10 ];  // ważne jest że i+3 % 10 jest translacją
        arr[i].data = i;
    }
    arr[7].next = nullptr; //dlaczego?
```

Do wyświetlenia zgodnie z numeracją można teraz użyć pętli `for`, a do wyswietlenia zgodnie z połączeniami poprzedniego rozwiązania (lub pętli `for`, jak?).

Patrz folder `zadanie_2`.

Wskazówka do zadania 3.

UWAGA! Uzyty algorytm zakłada, że ciag węzłów się kończy, tzn jest węzeł, który wskazuje na `nullptr`. Jeżeli tak nie będzie, to program wpadnie w nieskończoną pętlę!

Do wyswietlenia listy połączonych węzłów można użyć poniższej funkcji, przechodzącej po kolejnych węzłach aż do napotkania `nullptr`, czyli końca listy.

```
write_list_cout(AiSD::SLLNode* head){
    AiSD::SLLNode* current = head;  // niepotrzebne bo head można bez szkody
                                    // iterować head = head->next
                                    // można zatem zamienić current na head

    while(current != nullptr){
        std::cout << current->data << " -> ";
        current = current->next;
    }
    std::cout << "NIL";
}

// ogólniejsza wersja powyższego, wymaga dołaczenia nagłówka `ostream` do skompilowania
write_list_stream(std::ostream& out, AiSD::SLLNode* head){
    AiSD::SLLNode* current = head;  // niepotrzebne bo head można bez szkody
                                    // iterować head = head->next
                                    // można zatem zamienić current na head

    while(current != nullptr){
        out << current->data << " -> ";
        current = current->next;
    }
    out << "NIL";

    return out;
}
```

Rozwiązanie można również oprzeć na przeciążonym operatorze <<

```
std::ostream& operator<<(std::ostream& out, AiSD::SLLNode* head){

    while(head != nullptr){
        out << head->data << " -> ";
        head = head->next;
    }
    out << "NIL";

    return out;
}
...
// tworzenie węzłów jak w poprzednim zadaniu
...
SLLNode* head = n0;
    std::cout << head;
```

Uwaga użycie funkcji `operatora<<` w ten sposób, jest niekoniecznie dobrym rozwiazaniem, gdyż użytkownik mógły oczekiwać raczej wypisania zawartości jednego węzła a nie całej listy.

W przypadku listy dwukierunkowej wystarczy zmodyfikować powyższe.

Ponieważ nie ma prostego sposobu (chyba) aby przeciążyć operator << tak aby ten sam typ danych wyświetlał na dwa sposoby więc wyświetlanie w kierunku wyznaczonym przez `prev` najłatwiej jest zrealizować przez zdefiniowanie funkcji `write_to_???_reversed(...)`. Jest to kolejny argument za

nieprzeciążaniem tego operatora.

Patrz folder `zadanie_3`.

Wskazówka do zadania 4.

Coś pusi pozostać dla Państwa.