Podstawy Programowania 2

Dwukierunkowa lista cykliczna

Arkadiusz Chrobot

Zakład Informatyki

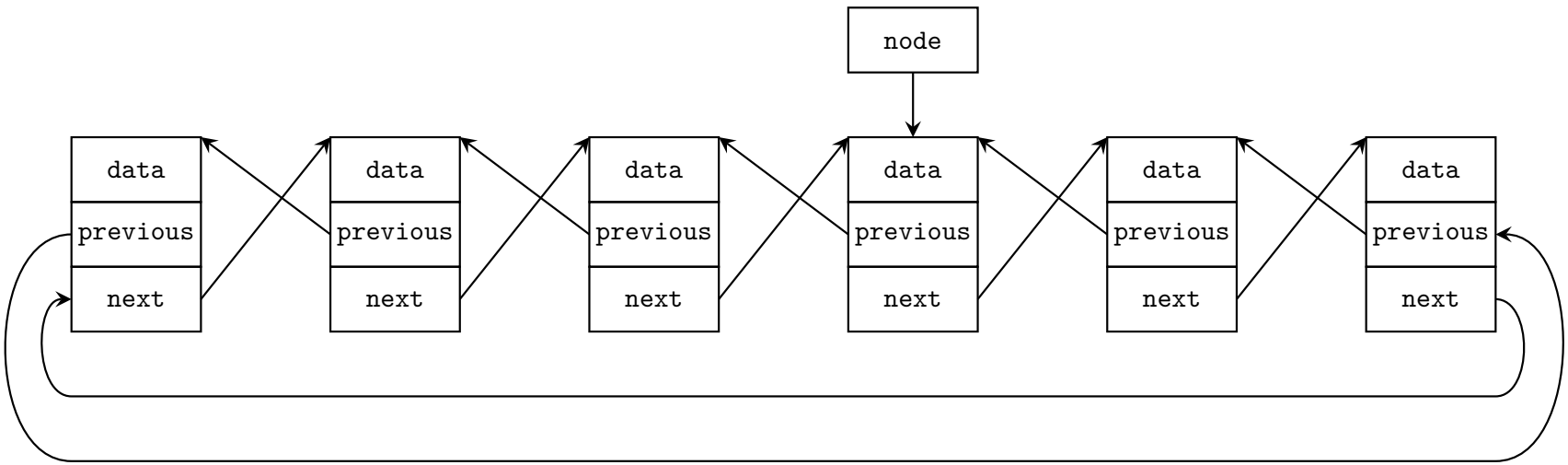
15 kwietnia 2019

Plan

Wstęp

Lista cykliczna (ang. *Circular Linked List*) jest listą w której nie jest wyróżniony element początkowy, ani końcowy. Każdy element ma swojego następcę i poprzednika. Listy takie można konstruować jako jedno lub dwukierunkowe. Liniowa lista jednokierunkowa może zostać przekształcona w listę cykliczną poprzez zapisanie w polu wskaźnikowym jej ostatniego elementu adresu elementu pierwszego. Podobne operacje pozwalają zamienić dwukierunkową listę liniową w listę cykliczną. W ramach wykładu zaprezentowany zostanie program, który tworzy listę, która od momentu powstania jest dwukierunkową listą cykliczną. Następny slajd zawiera rysunek przedstawiający schematycznie taką listę.

Wstęp



Dwukierunkowa lista cykliczna

Implementacja

Podobnie jak w przypadku poprzednich list, dwukierunkowa lista cykliczna zostanie zaprezentowana przy pomocy programu, który przechowuje w niej liczby naturalne uporządkowane niemalejąco. Ponieważ lista cykliczna nie ma początku ani końca to ten porządek będzie względny, tzn. zależny od elementu przechowującego bieżąco najmniejszą z liczb zapisanych w liście.

Lista cykliczna może być zrealizowana przy pomocy tablicy lub jako dynamiczna struktura danych. Na wykładzie będzie interesował nas ten drugi sposób jej tworzenia. Takie listy dosyć często są realizowane jako listy z wartownikiem, co upraszcza implementację operacji na nich. Zaprezentowane rozwiązanie nie zawiera takiego udogodnienia.

Typ bazowy i wskaźnik listy

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  const unsigned int how_many = 2;
5
6  struct list_node
7  {
8      int data;
9      struct list_node *previous, *next;
10 } *list_pointer;
```

Typ bazowy

Typ bazowy dwukierunkowej listy cyklicznej jest taki sam, jak typ bazowy dwukierunkowej listy liniowej. Można go również poddawać takim samym modyfikacjom, zależnie od problemu jaki staramy się rozwiązać przy użyciu takiej listy. Ze względu na konstrukcję dwukierunkowej listy cyklicznej nie będzie ona zawierała elementu, którego oba pola wskaźnikowe lub jedno z tych pól miałoby wartość `NULL`.
Oprócz definicji typu bazowego listy i deklaracji jej wskaźnika, na poprzednim slajdzie znajdują się instrukcje preprocesora włączające do programu te same pliki nagłówkowe, co w przypadku innych programów dotyczących list oraz definicja stałej, która będzie używana jako argument funkcji wypisującej wartości elementów listy na ekran.

Wskaźnik listy

W przypadku listy cyklicznej, obojętnie czy jedno, czy dwukierunkowej, wskaźnik listy może wskazywać na dowolny element należący do niej. Nie ma żadnych wymogów, odnośnie tego, który z nich to będzie. Jeśli ten wskaźnik jest pusty (ma wartość `NULL`), to oznacza to, że lista jest pusta.

Operacje na liście

Dwukierunkowa lista cykliczna jest, podobnie jak poprzednio opisywane listy, abstrakcyjną strukturą danych. Oprócz określenia jej typu bazowego musimy zatem zdefiniować operacje, jakie będą na niej przeprowadzane. Tak jak poprzednio ograniczymy się do tych najbardziej podstawowych: tworzenia listy, dodawania elementu (wraz z wyszukiwaniem elementu w liście), usuwania elementu, wypisywania elementów oraz usuwania listy.

Tworzenie listy

```
1  struct list_node *create_list(int number)
2  {
3      struct list_node *new_node = (struct list_node *)
4                                   malloc(sizeof(struct list_node));
5      if(new_node) {
6          new_node->data = number;
7          new_node->previous = new_node->next = new_node;
8      }
9      return new_node;
10 }
```

Tworzenie listy

Przedstawiona na poprzednim slajdzie funkcja tworzy dwukierunkową listę cykliczną poprzez utworzenie jej pierwszego elementu. Budowa tej funkcji w dużej mierze jest taka sama jak analogicznej funkcji dla dwukierunkowej listy liniowej. Różnica polega na zainicjowaniu pól wskaźnikowych tworzonego elementu nie wartością `NULL`, ale adresem tego elementu (wiersz nr 7). Dzięki temu powstaje dwukierunkowa lista cykliczna składająca się tylko z jednego elementu.

Dodawanie elementu do listy

Operacja dodawania elementu do dwukierunkowej listy cyklicznej musi być przeprowadzana dla niepustej listy. Tak, jak w przypadku poprzednich list chcemy, aby wartości w elementach dwukierunkowej listy cyklicznej tworzyły uporządkowany niemalejąco ciąg. Problem znajdywania miejsca dla nowego elementu w takiej liście sprowadza się do znalezienia elementu o wartości większej od tej, która jest w nim zapisana lub do znalezienia elementu o wartości minimalnej w całej liście. Ten ostatni przypadek ma zastosowanie w sytuacji wstawiania elementu o wartości większej od wartości wszystkich elementów dotychczas tworzących listę. Obie operacje wyszukiwania są zaimplementowane w osobnych, pomocniczych funkcjach, które będą przedstawione na kolejnych slajdach.

Dodawanie elementu do listy

Znajdowanie najmniejszej wartości w liście

```
1  struct list_node *find_minimum_value_node(
2                                       struct list_node *list_pointer)
3  {
4          struct list_node *start, *result;
5          start = result = list_pointer;
6          int minimum = list_pointer->data;
7          do  {
8                  if(minimum>list_pointer->data) {
9                      minimum = list_pointer->data;
10                     result = list_pointer;
11                 }
12                 list_pointer = list_pointer->next;
13         } while(list_pointer!=start);
14         return result;
15 }
```

Dodawanie elementu do listy

Znajdowanie najmniejszej wartości w liście

Zadaniem przedstawionej na poprzednim slajdzie funkcji jest odnalezienie elementu listy zawierającego minimalną liczbę zapisaną w tej liście. W przypadku gdyby taka wartość kilkukrotnie się powtarzała, czyli na liście występowałoby skupisko elementów, inaczej *klaster*, zawierających te same wartości, to ta funkcja znajduje pierwszy element tego klastra. Funkcja `find_minimum()` używa do znalezienia elementu o wartości minimalnej modyfikacji algorytmu, który wyszukuje wartość minimalną w nieuporządkowanej tablicy. Do tej funkcji przekazywany jest wskaźnik na listę, a zwraca ona adres elementu zawierającego liczbę o minimalnej wartości. Algorytm jej działania wymaga, aby wszystkie elementy listy zostały sprawdzone w pętli. Ponieważ w żadnym polu wskaźnikowym elementów listy nie ma zapisanej wartości `NULL`, która sygnalizowałaby kiedy pętla musi się zatrzymać, to należy rozwiązać problem jej zakończenia.

Dodawanie elementu do listy

Znajdowanie najmniejszej wartości w liście

W wierszu nr 4 funkcji `find_minimum_value_node()` zadeklarowane są dwie zmienne wskaźnikowe: `start` i `result`. W kolejnym wierszu obie te zmienne są inicjowane bieżącą wartością wskaźnika listy, ale ich rola będzie inna. Zmienna `start` będzie przechowywała adres elementu, od którego rozpocznie się przeszukiwanie listy. Do „poruszania się" po liście będzie służył w pętli wskaźnik `list_pointer`. Jeśli jego wartość pokryje się z wartością zmiennej `start`, to będzie to oznaczało, że wszystkie elementy listy zostały odwiedzone i należy ją zakończyć. Aby pętla przynajmniej raz się wykonała ten warunek powinien być badany po wykonaniu instrukcji stanowiących jej ciało, dlatego zamiast pętli `while` została zastosowana w tej funkcji pętla `do...while`. Po zakończeniu tej pętli zmienna `result` będzie wskazywała element o najmniejszej wartości w liście i jego adres zostanie zwrócony przez funkcję (wiersz nr 14). Zmienna lokalna `minimum` będzie używana do zapamiętania tej wartości. Obie zmienne są używane w pętli.

Dodawanie elementu do listy

Znajdowanie najmniejszej wartości w liście

Najpierw sprawdzane jest w niej, czy bieżący element ma wartość mniejszą od tej zapamiętanej bieżąco w `minimum` (wiersz nr 8). Jeśli tak, to jego wartość zapamiętywana jest w tej zmiennej, a w zmiennej `result` jego adres. Po zakończeniu pętli `result` zawiera adres szukanego elementu.

Dodawanie elementu do listy

Znajdywanie miejsca dla elementu

```
1  struct list_node *find_next_node(struct list_node *list_pointer,
2                                                    int number)
3  {
4          list_pointer = find_minimum_value_node(list_pointer);
5          struct list_node *start = list_pointer;
6          do {
7              if(list_pointer->data>number)
8                  break;
9              list_pointer = list_pointer->next;
10         } while(list_pointer!=start);
11         return list_pointer;
12 }
```

Dodawanie elementu do listy

Znajdywanie miejsca dla elementu

Funkcja `find_next_node()` zwraca adres elementu, przed którym należy wstawić nowy element do listy. Przyjmuje ona dwa argumenty wywołania, czyli wskaźnik listy oraz liczbę zapisaną w nowym elemencie. Najpierw wywołuje ona `find_minimum_value_node()` celem znalezienia elementu listy o najmniejszej wartości. Będzie to punkt startowy dla dalszych poszukiwań (wiersz nr 5), które będą przeprowadzane w pętli `do…while`. Sposób określenia warunku zatrzymania tej pętli jest taki sam, jak w poprzednio opisanej funkcji - pętla wykonuje się tak długo, aż wskaźnik listy „wróci” do elementu, od którego zaczęło się jej wykonanie (wiersz nr 10). Możliwy jest również inny scenariusz jej zakończenia. Podczas jej wykonania może zostać napotkany element o większej wartości od tej, która będzie umieszczona w nowym elemencie (wiersz nr 7). Jeśli tak się stanie to wykonanie pętli zostanie przerwane (wiersz nr 8). Niezależnie od sposobu zatrzymania pętli, wskaźnik listy będzie zawierał adres elementu, przed którym należy wstawić nowy i on zostanie przez funkcję zwrócony (wiersz nr 11).

Dodawanie elementu do listy

Dodanie elemenut do listy

```
1  void add_node(struct list_node *list_pointer, int number)
2  {
3      if(list_pointer) {
4          struct list_node *new_node = (struct list_node *)
5                                      malloc(sizeof(struct list_node));
6          if(new_node) {
7                  new_node->data = number;
8                  list_pointer =
9                            find_next_node(list_pointer,number);
10                 new_node->next = list_pointer;
11                 new_node->previous = list_pointer->previous;
12                 list_pointer->previous->next = new_node;
13                 list_pointer->previous=new_node;
14         }
15     }
16 }
```

Dodawanie elementu do listy

Funkcja `add_node()` wykonuje dodanie elementu do listy. Przyjmuje ona dwa argumenty wywołania, czyli wskaźnik na listę i liczbę, którą należy zapisać w jej nowym elemencie. Ponieważ wynik działania funkcji jest widoczny po wypisaniu zwartości listy na ekranie, to nie zwraca ona żadnej wartości. Najpierw sprawdza ona, czy lista, do której element ma być dodany istnieje (wiersz nr 3). Jeśli tak, to przydziela pamięć na ten element (wiersze 4 i 5). Jeśli ten przydział się nie powiedzie, to funkcja kończy działanie, a stan listy pozostaje niezmieniony. Jeśli jednak przydział zakończy się sukcesem (wiersz nr 6), to w nowym elemencie zapisywana jest przekazana do funkcji liczba (wiersz nr 7), a następnie funkcja ustala adres elementu, przed którym należy wstawić nowy. W tym celu wywołuje `find_next_node()` (wiersze nr 8 i 9). Po jej zakończeniu funkcja `add_node()` dołącza nowy element do listy (wiersze nr 10, 11, 12 i 13). Odbywa to się w podobny sposób jak dodanie nowego elementu wewnątrz dwukierunkowej listy liniowej.

Usuwanie elementu z listy

Implementując operację usuwania elementu z dwukierunkowej listy cyklicznej należy rozważyć dwie sytuacje:

1. usuwany jest element z jednoelementowej listy cyklicznej,
2. usuwany jest element z listy zawierającej więcej niż jeden element.

W pierwszym przypadku po usunięciu elementu lista stanie się pusta, w drugim lista stanie się tylko mniejsza o jeden element. Należy również zadbać, aby ta operacja była przeprowadzana na niepustej liście. Stan listy nie będzie ulegał zmianie tylko wtedy, gdy jest ona pusta lub nie zawiera elementu, który należałoby usunąć.

Usuwanie z listy

```
1  struct list_node *delete_node(struct list_node *list_pointer, int number)
2  {
3      if(list_pointer) {
4          list_pointer = find_next_node(list_pointer,number);
5          list_pointer = list_pointer->previous;
6          if(list_pointer->data == number) {
7              if(list_pointer == list_pointer->next) {
8                  free(list_pointer);
9                  return NULL;
10             }
11             struct list_node *next = list_pointer->next;
12             list_pointer->previous->next = list_pointer->next;
13             list_pointer->next->previous = list_pointer->previous;
14             free(list_pointer);
15             list_pointer=next;
16         }
17     }
18     return list_pointer;
19 }
```

Usuwanie z listy

Funkcja `delete_node()` przyjmuje dwa argumenty wywołania: wskaźnika na listę oraz liczbę, którą powinien zawierać element do usunięcia. W przypadku kiedy jest wiele elementów zawierających taką liczbę, to wystarczy, że zostanie usunięty jeden z nich. Funkcja zwraca wskaźnik na dowolny element listy, jeśli po usunięciu elementu lista zawiera inne elementy lub `NULL`, jeśli po wykonaniu tej operacji lista staje się pusta. W wierszu nr 3 `delete_node()` najpierw sprawdza, czy została wywołana dla listy, która nie jest pusta. Jeśli tak jest to ustala ona położenie elementu, który należy usunąć. W tym celu wywołuje funkcję `find_next_node()` (wiersz nr 4), ale ta zwraca wskaźnik na element o wartości większej niż poszukiwana. Zatem w wierszu nr 5 funkcja `delete_node()` cofa wskaźnik `list_pointer` o jeden element, a następnie w wierszu nr 6 sprawdza, czy ten element zawiera poszukiwaną wartość. W wierszu nr 7 sprawdzane jest dodatkowo, czy nie jest to jedyny element tej listy. Aby uzyskać odpowiedź na to ostatnie pytanie należy sprawdzić, czy dowolne z pól wskaźnikowych elementu nie wskazuje na niego. W tym przypadku wybór padł na `next`.

Usuwanie z listy

Jeśli warunek z wiersza nr 7 jest prawdziwy, to oznacza, że należy usunąć jedyny element listy. Zatem w wierszu nr 8 zwalniana jest pamięć przeznaczona na ten element, a w wierszu nr 9 funkcja zwraca wartość `NULL` i kończy swoje wykonanie. Jeśli warunek z wiersza nr 7 nie jest spełniony, to funkcja usuwa element z listy zawierającej co najmniej dwa elementy. Zatem najpierw zapamiętuje ona adres następnego elementu względem tego usuwanego we wskaźniku lokalnym `next` (wiersz 12), a potem wyłącza usuwany element z listy (wiersze 13 i 14). Operacja ta przebiega identycznie, jak w przypadku usuwania elementu z wnętrza dwukierunkowej listy liniowej. Następnie zwalniana jest pamięć przeznaczona na usuwany element (wiersz nr 15) i wskaźnikowi listy przypisywany jest adres zapamiętany w zmiennej `next` (wiersz nr 16). To przypisanie jest potrzebne, ponieważ w wierszu nr 21 funkcja zwraca ten wskaźnik i musi on wskazywać na istniejący element listy.

Usuwanie z listy

Gdyby lista przekazana do funkcji `delete_node()` była pusta, to zwróciłaby ona w wierszu nr 21 wartość `NULL`. Gdyby natomiast okazało się, że nie ma w liście elementu, który należałoby usunąć, to zwróciłaby ona tę samą wartość wskaźnika listy, jaka została jej przekazana podczas wywołania.

Wypisywanie zawartości listy

```
1  void print_list(struct list_node *list_pointer,
2                                        const unsigned int how_many)
3  {
4      if(list_pointer) {
5          list_pointer = find_minimum_value_node(list_pointer);
6          int i;
7          for(i=0; i<how_many; i++) {
8              struct list_node *start = list_pointer;
9              do {
10                 printf("%d ",list_pointer->data);
11                 list_pointer = list_pointer->next;
12             } while(list_pointer!=start);
13             puts("");
14         }
15     }
16 }
```

Wypisywanie zawartości listy

Operacja wypisywania elementów z listy została zaimplementowana w postaci funkcji `print_list()`. Funkcja ta nie zwraca żadnej wartości, ale przyjmuje dwa argumenty wywołania. Pierwszym jest wskaźnik na listę, a drugim liczba określająca ile razy (w osobnych wierszach) ma być wypisana zawartość listy. Parametr, przez który jest przekazywany ten argument nazywa się `how_many`. Argumentem, który jest za niego podstawiany jest stała, która ma taką samą nazwę, a jej wartość wynosi 2. Po sprawdzeniu, czy lista, która ma być wypisana istnieje (wiersz nr 4) funkcja ustala adres elementu o najmniejszej wartości za pomocą wywołania `find_minimum_value_node()`. Nie jest to konieczne, ale ułatwia sprawdzenie, że faktycznie liczby zapisane w liście tworzą ciąg niemalejący. Następnie w pętli `for` funkcja zapamiętuje w zmiennej lokalnej `start` adres elementu, od którego rozpocznie wypisywanie wartości elementów na ekranie i wypisuje je używając do tego pętli `do...while`. Po zakończeniu tej pętli kursor jest przenoszony do następnego wiersza ekranu za pomocą wywołania `puts()` i zależnie od wartości parametru `how_many` rozpoczyna się kolejna iteracja pętli `for`.

Usuwanie listy

```
1  void remove_list(struct list_node **list_pointer)
2  {
3      if(*list_pointer) {
4          struct list_node *start = *list_pointer;
5          do {
6              struct list_node *next = (*list_pointer)->next;
7              free(*list_pointer);
8              *list_pointer = next;
9          } while(*list_pointer!=start);
10         *list_pointer = NULL;
11     }
12 }
```

Usuwanie listy

Usuwanie listy zaimplementowane jest podobnie jak w przypadku liniowych list jedno i dwukierunkowych. Funkcja `remove_list()` różni się od analogicznych funkcji dla wspomnianych list wstępnym sprawdzaniem, czy usuwana lista istnieje (wiersz nr 3), konstrukcją i rodzajem użytej pętli, w której usuwane są poszczególne elementy oraz tym, że po jej zakończeniu wskaźnikowi listy jest przypisywana wartość `NULL` (wiersz nr 10). Ta ostatnia czynność podyktowana jest tym, że po opróżnieniu listy wskaźnik na nią musi być pusty. Ponieważ nie ma w liście pola wskaźnikowego, które posiadałoby wartość `NULL`, to nie gwarantuje tego instrukcja z wiersza nr 8. Pętla użyta do zwalniania pamięci przeznaczonej na poszczególne elementy listy to `do…while`. Jej sposób użycia jest podobny jak we wcześniej zaprezentowanych funkcjach. Po pobieżnej analizie możemy mieć wątpliwości, czy sprawdzenie warunku w wierszu nr 9 jest bezpieczne, gdyż wskaźnik `start` zawiera adres elementu, który został usunięty. Odpowiedź jest pozytywna. Sprawdzany jest tylko adres w tym wskaźniku, nie jest wykonywana jego dereferencja, a więc nie jest odczytywana zawartość nieistniejącego elementu.

Funkcja main()

Część pierwsza

```
1  int main(void)
2  {
3      list_pointer = create_list(1);
4      int i;
5      for(i=2;i<5;i++)
6          add_node(list_pointer,i);
7      for(i=6;i<10;i++)
8          add_node(list_pointer,i);
9      print_list(list_pointer,how_many);
```

Funkcja `main()`

Część pierwsza

W pierwszej części funkcji `main()` tworzona jest dwukierunkowa lista cykliczna z jednym elementem o wartości `1` (wiersz nr 3), następnie (podobnie jak w przypadku poprzednio opisywanych list) dodawane są do niej elementy o wartościach od 2 do 4 i od 6 do 9 (wiersze nr 5 i 6 oraz 7 i 8). Potem zawartość listy jest wypisywana na ekranie dwukrotnie, zgodnie z wartością stałej `how_many` (wiersz nr 9). Zwiększając lub zmniejszając wartość tej stałej przed kompilacją możemy wpływać na to ile razy zawartość listy będzie wypisana przy pojedynczym wywołaniu funkcji `print_list()`.

Funkcja `main()`

Część druga

```
1       add_node(list_pointer,0);
2       print_list(list_pointer,how_many);
3       add_node(list_pointer,5);
4       print_list(list_pointer,how_many);
5       add_node(list_pointer,7);
6       print_list(list_pointer,how_many);
7       add_node(list_pointer,10);
8       print_list(list_pointer,how_many);
```

Funkcja `main()`

Część druga

W drugiej części funkcji `main()` do listy są dodawane elementy o wartościach `0`, `5`, `7` i `10`. Po każdym dodaniu zawartość listy jest wypisywana dwukrotnie na ekranie.

Funkcja main()

Część trzecia

```
1       list_pointer = delete_node(list_pointer,0);
2       print_list(list_pointer,how_many);
3       list_pointer = delete_node(list_pointer,1);
4       print_list(list_pointer,how_many);
5       list_pointer = delete_node(list_pointer,1);
6       print_list(list_pointer,how_many);
7       list_pointer = delete_node(list_pointer,5);
8       print_list(list_pointer,how_many);
9       list_pointer = delete_node(list_pointer,7);
10      print_list(list_pointer,how_many);
11      list_pointer = delete_node(list_pointer,10);
12      print_list(list_pointer,how_many);
13      remove_list(&list_pointer);
14      return 0;
15  }
```

Funkcja `main()`

Część trzecia

W trzeciej części funkcji `main()` z listy usuwane są elementu o wartościach `0`, `1`, ponownie `1` (tym razem nie zostanie usunięty żaden element), `5`, `7` (zostanie usunięty tylko jeden z dwóch elementów zawierających taką wartość) oraz `10`. Po każdej takiej operacji zawartość listy jest wypisywana na ekranie. Ostatnią operacją wykonywaną na liście w funkcji `main()` jest usunięcie tej listy. Potem funkcja zwraca `0` i kończy swoje działanie. Czynności wykonywane w funkcji `main()` nie wyczerpują możliwości testowania tej implementacji dwukierunkowej listy cyklicznej, ale stanowią podstawowe minimum pozwalające sprawdzić poprawność działania najważniejszych operacji.

Podsumowanie

Przedstawiona implementacja nie wykorzystuje wszystkich możliwości, jakie daje dwukierunkowa lista cykliczna, np. funkcja `print_list()` wypisuje zawartość listy tylko w jednym kierunku. Podobnie jak listy liniowe, listy cykliczne mogą być zrealizowane z użyciem tablicy, lub tak jak wspomniano wcześniej posiadać element będący wartownikiem. Tego typu listy mają zastosowanie np. w systemach operacyjnych w realizacji rotacyjnego algorytmu szeregowania lub jako bufory np. pakietów danych obsługiwanych przez podsystemy komunikacji sieciowej. D.E.Knuth w pierwszej części „Sztuki programowania" opisuje algorytm, w którym listy cykliczne są używane do mnożenia wielomianów. Poszczególne elementy takich list zawierają ich współczynniki.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!