Podstawy Programowania 2

Jednokierunkowa lista liniowa

Arkadiusz Chrobot

Zakład Informatyki

24 marca 2020

Plan

Jednokierunkowa lista liniowa

Jednokierunkowa lista liniowa jest abstrakcyjną strukturą danych, która może przechowywać dane zarówno w sposób uporządkowany, jak i nieuporządkowany. W odróżnieniu od tablicy pozwala ona jedynie na sekwencyjny dostęp do zgromadzonych w niej danych, ale nowe informacje mogą być w niej umieszczane w dowolnym jej miejscu. Taką samą cechę ma także operacja usuwania danych z listy. Istnieje kilka odmian list. Stosy i kolejki są szczególnymi przypadkami listy. Jednokierunkowa lista liniowa wyróżnia się spośród pozostałych list tym, że można w niej wyróżnić element początkowy i końcowy oraz, że (przynajmniej w założeniu) można przeglądać jej elementy tylko w jedną stronę.

Jednokierunkowa lista liniowa

Jednokierunkowa lista liniowa może być zrealizowana jako dynamiczna struktura danych i takiej jej implementacji poświęcony będzie ten wykład. Można ją także zbudować w oparciu o tablicę. Ta możliwość zostanie opisana krótko pod koniec wykładu. Aby zrealizować listę w postaci dynamicznej struktury danych należy określić typ pojedynczego jej elementu, inaczej typ bazowy listy oraz operacje, jakie będą na tej liście wykonywane. W przykładowym programie ograniczymy się jednie do pięciu podstawowych operacji: tworzenia listy, wstawiania nowego elementu, usuwania elementu z listy, wypisywania zwartości listy na ekranie i usuwania całej listy z pamięci. Zostanie także, w pewnej postaci, zrealizowana operacja wyszukiwania elementu o zadanej wartości.

Implementacja

Konstrukcja jednokierunkowej listy liniowej zostanie przedstawiona na przykładzie programu, który przechowuje w niej liczby naturalne, choć implementacja listy będzie przewidywała także możliwość umieszczania w niej liczb całkowitych. Dodatkowo, liczby na tej liście będą zawsze posortowane niemalejąco.

Typ bazowy i wskaźnik listy

```c
#include<stdio.h>
#include<stdlib.h>

struct list_node {
    int data;
    struct list_node *next;
} *list_pointer;
```

Typ bazowy i wskaźnik listy

W programie zostaną użyte te same pliki nagłówkowe, co w przypadku kolejki prezentowanej na poprzednim wykładzie. Również typ bazowy jednokierunkowej listy liniowej jest, z dokładnością do nazwy, taki sam jak w przypadku stosu i kolejek. Podobnie jak w przypadku tych struktur można go modyfikować zmieniając i dodając pola do przechowywania danych. Można także dodawać nowe pola wskaźnikowe, ale zawsze musi w nim być zawarte co najmniej jedno takie pole, które może wskazywać na elementy tego samego typu co typ bazowy. Służy ono do tworzenia struktury listy, czyli łączenia ze sobą elementów. Warto zauważyć, że to pole w ostatnim elemencie listy będzie zawsze miało wartość `NULL`. W prezentowanym programie typ bazowy będzie zawierał jedno pole służące do przechowywania liczb typu `int` oraz jedno pole wskaźnikowe. Definicję typu bazowego połączono z deklaracją wskaźnika na listę (wiersz 7). Będzie on zatem zmienną globalną. Aby operacje na liście przebiegały w sposób prawidłowy, bez „gubienia" jej elementów będziemy wymagać, aby ten wskaźnik zawsze zawierał adres pierwszego elementu listy.

Tworzenie listy

Operacja tworzenia listy, czyli dodania do niej pierwszego elementu może być zrealizowana na wiele sposobów. W prezentowanym programie została ona wyodrębniona do osobnego podprogramu, którego kod źródłowy jest zamieszczony na następnym slajdzie.

Tworzenie listy

Funkcja create_list()

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

Tworzenie listy

Funkcja `create_list()`

Funkcja tworząca pierwszy element listy przyjmuje przez parametr daną, która ma być w nim umieszczona, a zwraca jego adres, jeśli operacja utworzenia elementu listy zakończy się powodzeniem, lub wartość `NULL` w przeciwnym przypadku. W wierszach 3 i 4 tej funkcji następuje przydział pamięci dla nowego elementu. Jeśli ten przydział zakończy się pomyślnie, to funkcja wykonuje inicjację pól i zwraca adres tego elementu. Jeśli pamięć nie zostanie przydzielona, to element nie będzie utworzony, i zostanie zwrócona wartość `NULL` zawarta w zmiennej `first`. Proszę zwrócić uwagę, że pole `next` pierwszego elementu jest inicjowane także wartością `NULL`, ponieważ jest to pierwszy i zarazem ostatni element tej listy. Wartość zwrócona przez funkcję `create_list()` będzie przypisana wskaźnikowi listy.

Operacja wstawiania elementu do listy

Zakładamy, że operacja wstawiania nowego elementu do listy będzie wykonywana zawsze na niepustej liście, czyli takiej, która zawiera co najmniej jeden istniejący element. Dodatkowo, będziemy wymagać, aby wskaźnik listy zarówno przed, jak i po wykonaniu tej operacji wskazywał na pierwszy element należący do listy oraz aby wartości jej elementów były uporządkowane niemalejąco. Jeśli utworzenie nowego elementu się nie powiedzie, to operacja wstawienia elementu powinna pozostawić listę w takim samym stanie, w jakim ona była przed jej rozpoczęciem.

Operacja wstawiania elementu do listy

Implementacja

Jeśli lista ma przechowywać liczby w sposób uporządkowany niemalejąco, to należy rozważyć i poprawnie oprogramować trzy możliwości wstawienia nowego elementu do niej:

1. elementu będzie dodany na początku listy; po dodaniu będzie jej pierwszym elementem,
2. element będzie dodany wewnątrz listy,
3. element będzie dodany na końcu listy; po dodaniu będzie jej ostatnim elementem.

Dodawaniem elementu do listy będzie zajmowała się pojedyncza funkcja, ale będzie ona wywoływała funkcje pomocnicze realizujące każdy z wymienionych przypadków. **Uwaga! Aby ułatwić zrozumienie implementacji operacji wstawiania nowego elementu do listy, wszystkie funkcje związane z jej realizacją zostaną zaprezentowane i opisane w odwrotnym porządku, niż zostały zdefiniowane w programie.** Kod źródłowy programu jest do pobrania ze strony internetowej przedmiotu.

Operacja wstawiania elementu do listy

Funkcja add_node()

```
1  struct list_node *add_node(struct list_node *list_pointer, int data)
2  {
3      struct list_node *new_node = (struct list_node *)
4                                          malloc(sizeof(struct list_node));
5      if(list_pointer && new_node) {
6          new_node->data = data;
7          if(list_pointer->data>=data) {
8              return add_at_front(list_pointer, new_node);
9          } else {
10             struct list_node *node= find_spot(list_pointer,data);
11             add_in_middle_or_at_back(node,new_node);
12         }
13     }
14     return list_pointer;
15 }
```

Operacja wstawiania elementu do listy

Funkcja `add_node()`

Funkcja `add_node()` jest wspomnianą wcześniej funkcją wstawiającą nowy element do listy. Przyjmuje dwa argumenty wywołania. Pierwszym jest wskaźnik na listę, a drugim liczba, która ma być do listy zapisana. Wartością zwracaną przez tę funkcję jest wskaźnik na pierwszy element listy. Jest to przydatną cechą w przypadku wstawiania nowego elementu na początku listy. W pozostałych przypadkach funkcja zwraca ten sam adres, który został jej przekazany przez parametr `list_pointer`. Wynik funkcji powinien być przypisany do wskaźnika listy w miejscu jej wywołania. Funkcja `add_node()` przydziela pamięć na nowy element, a następnie sprawdza, czy ta operacja się udała i czy lista, do której ma być wstawiony ten element istnieje (wiersz 5). Proszę zwrócić uwagę, że taka forma sprawdzenia mogłaby prowadzić do potencjalnych wycieków pamięci, jeśli element udałoby się utworzyć, ale nie istniałaby lista. W takim przypadku funkcja zwróciłaby wartość `NULL`, ale nie dodałaby utworzonego elementu do listy i adres do niego zostałby zgubiony.

Operacja wstawiania elementu do listy

Funkcja `add_node()`

W opisywanym programie taka sytuacja najczęściej nie zachodzi, bo ta funkcja jest zawsze wywoływana po `create_list()`. Jeśli jednak chcielibyśmy użyć `add_node()` w innym programie, to musimy zadbać, aby zawsze był jej przekazywany wskaźnik na niepustą listę. Jeśli lista i nowy element istnieją, to funkcja inicjuje pole `data` nowego elementu (wiersz nr 6). Inicjacja pola `next` nie jest konieczna, bo podczas kolejnych działań, w każdym z opisanych na wcześniejszym slajdzie przypadków, zostanie mu nadana prawidłowa wartość. Po inicjacji nowego elementu funkcja `add_node()` musi rozpoznać, który z nich wystąpił. Lista, do której element jest wstawiany jest uporządkowana, a więc jeśli nowy element ma być dodany na jej początku, to liczba w nim zapisana musi być mniejsza lub równa tej zapisanej w bieżącym pierwszym elemencie listy. Ten warunek jest sprawdzany w wierszu nr 7 funkcji. Jeśli jest on spełniony, to `add_node()` wywołuje funkcję `add_at_front()`, zwraca adres zwrócony przez tę ostatnią funkcję i kończy swoje działanie.

Operacja wstawiania elementu do listy

Funkcja `add_node()`

Jeśli warunek z wiesza nr 7 funkcji nie jest spełniony, to musi wystąpić, któryś z pozostałych dwóch przypadków: albo nowy element powinien być dodany wewnątrz listy, albo na jej końcu. Okazuje się, że oba te przypadki mogą zostać obsłużone w ten sam sposób. Najpierw jednak należy znaleźć element listy, *za którym* będzie dodany nowy. Tym zajmuje się funkcja `find_spot()`, która zwraca adres takiego elementu, który następnie jest zapisywany w lokalnej zmiennej wskaźnikowej `node`. Jeśli lista istnieje, co było sprawdzane wcześniej, to ta funkcja zawsze znajdzie odpowiedni element. Nie ma więc konieczności sprawdzania, czy wskaźnik `node` nie jest wskaźnikiem pustym. Po znalezieniu elementu za którym trzeba wstawić nowy, funkcja `add_node()` wywołuje funkcję `add_in_middle_or_at_back()`, która dokonuje wstawienia.

Wstawianie elementu na początku listy

Funkcja `add_at_front()`

```
1  struct list_node *add_at_front(struct list_node *list_pointer,
2                                 struct list_node *new_node)
3  {
4      new_node->next = list_pointer;
5      return new_node;
6  }
```

Wstawianie elementu na początku listy

Funkcja `add_at_front()`

Działanie funkcji `add_at_front()` jest zbliżone do działania funkcji `push()` znanej z wykładu o stosie. Funkcja ta przyjmuje dwa argumenty wywołania: wskaźnik na pierwszy element listy (tożsamy z wskaźnikiem na listę w przykładowym programie) i wskaźnik na nowy element listy. Ponieważ istnienie zarówno listy, jak i nowego elementu zostało potwierdzone w funkcji `add_node()`, to nie ma potrzeby dokonywania takiego sprawdzenia w opisywanej funkcji. Należy jednakże pamiętać, że nie powinna ona być używana poza `add_node()` bez sprawdzenia poprawności przekazywanych jej wskaźników. W wierszu nr 4 funkcja zapisuje w polu `next` nowego elementu wskaźnik na bieżący pierwszy element listy, a następnie (wiersz nr 5) zwraca adres nowego elementu (teraz jest to pierwszy element listy) i kończy swoje działanie.

Wstawianie elementu wewnątrz i na końcu listy

Funkcja `find_spot()`

```
1  struct list_node *find_spot(struct list_node *list_pointer,
2                                                      int data)
3  {
4      struct list_node *previous = NULL;
5      while(list_pointer && list_pointer->data<data) {
6          previous = list_pointer;
7          list_pointer = list_pointer->next;
8      }
9      return previous;
10 }
```

Wstawianie elementu wewnątrz i na końcu listy

Funkcja `find_spot()`

Funkcja `find_spot()` odpowiedzialna jest za znalezienie elementu listy **za** którym ma być wstawiony nowy element i zwrócenie wskaźnika na niego. Podobnie jak w przypadku funkcji `add_at_front()` nie jest konieczne sprawdzanie poprawności przekazanego do niej wskaźnika listy, gdyż jego poprawność została określona w funkcji `add_node()`. Dodatkowo do opisywanej funkcji przekazywana jest liczba, która zapisana jest w nowym elemencie. Do przeglądania elementów w pętli `while` funkcja będzie używała przekazanego jej przez wartość wskaźnika na listę. Dodatkowo wykorzystany zostanie w tej pętli wskaźnik `previous` zadeklarowany w wierszu nr 4 funkcji. Po pierwszej iteracji i każdej następnej będzie on wskazywał na element listy poprzedzający element wskazywany przez `list_pointer`. Wyjątkiem będzie sytuacja, w której ten ostatni wskaźnik osiągnie w wyniku wykonania pętli wartość `NULL`. Wówczas wskaźnik `previous` będzie wskazywał ostatni element listy.

Wstawianie elementu wewnątrz i na końcu listy

Funkcja `find_spot()`

Warunek pętli z wiersza nr 5 określa, że zakończy się ona, kiedy wskaźnik `list_pointer` będzie wskazywał element listy zawierający liczbę większą lub równą od tej, która będzie wstawiona w nowym elemencie, lub gdy osiągnie on wartość `NULL`. Ten ostatni przypadek oznacza, że dodawana do listy liczba będzie największa na niej i musi się znaleźć na jej końcu. Podsumowując: wskaźnik `list_pointer` po zakończeniu pętli `while` będzie wskazywał na element listy **przed** którym należy wstawić nowy lub będzie miał wartość `NULL`. Wskaźnik `previous` będzie wskazywał element **za** którym trzeba wstawić nowy i jego wartość będzie zwrócona przez funkcję.

Wstawianie elementu wewnątrz i na końcu listy

Funkcja `add_in_middle_or_at_back()`

```
1  void add_in_middle_or_at_back(struct list_node *node,
2                                struct list_node *new_node)
3  {
4      new_node->next = node->next;
5      node->next = new_node;
6  }
```

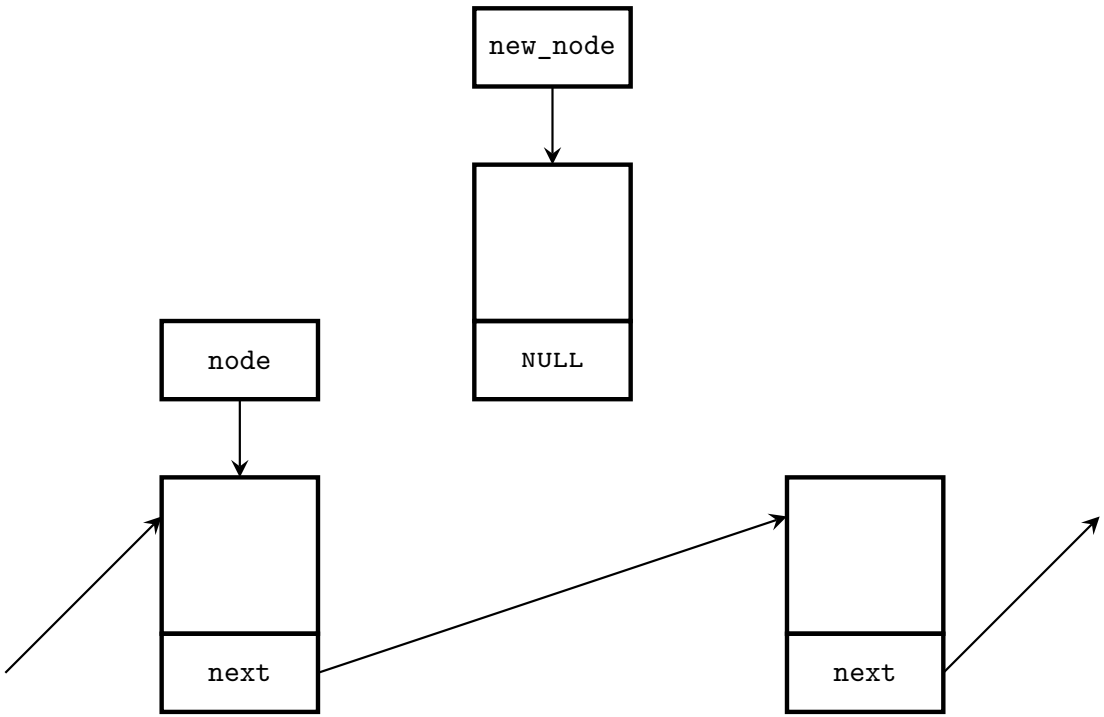
Wstawianie elementu wewnątrz i na końcu listy

Funkcja `add_in_middle_or_at_back()`

Zadaniem funkcji `add_in_middle_or_at_back()` jest włączenie nowego elementu do listy w jej wnętrzu lub na jej końcu. Po jej wykonaniu struktura listy musi być spójna, tzn. lista powinna być powiększona o jeden element i wszystkie jej elementy powinny być ze sobą prawidłowo powiązane. Do funkcji przez pierwszy parametr przekazywany jest wskaźniki na element listy, przed którym trzeba włączyć nowy. Wskaźnik na nowy element przekazywany jest przez drugi parametr. Ponieważ opisywana funkcja wywoływana jest z poziomu `add_node()`, to nie jest konieczne sprawdzanie poprawności tych wskaźników. W wierszu nr 4 funkcja `add_in_middle_or_at_back()` umieszcza w polu `next` nowego elementu adres elementu listy znajdującego się za elementem wskazywanym przez wskaźnik `node`. W wierszu nr 5 w polu `next` elementu wskazywanego przez `node` umieszczany jest adres nowego elementu. Po wykonaniu tych czynności nowy element staje się częścią listy. Wiersze nr 4 i 5 nie mogą być zamienione miejscami. Następne slajdy ilustrują opisane działanie funkcji.

Wstawianie elementu wewnątrz i na końcu listy

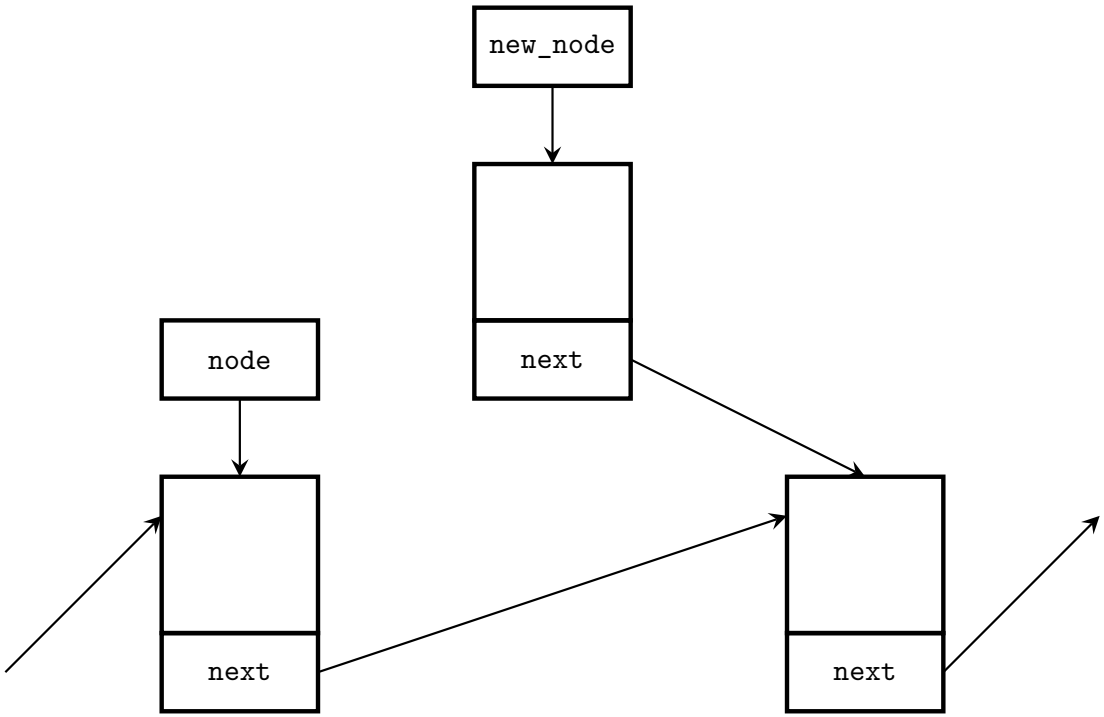
Funkcja `add_in_middle_or_at_back()`



Sytuacja przed wykonaniem wiersza nr 4 funkcji `add_in_middle_or_at_back()`

Wstawianie elementu wewnątrz i na końcu listy

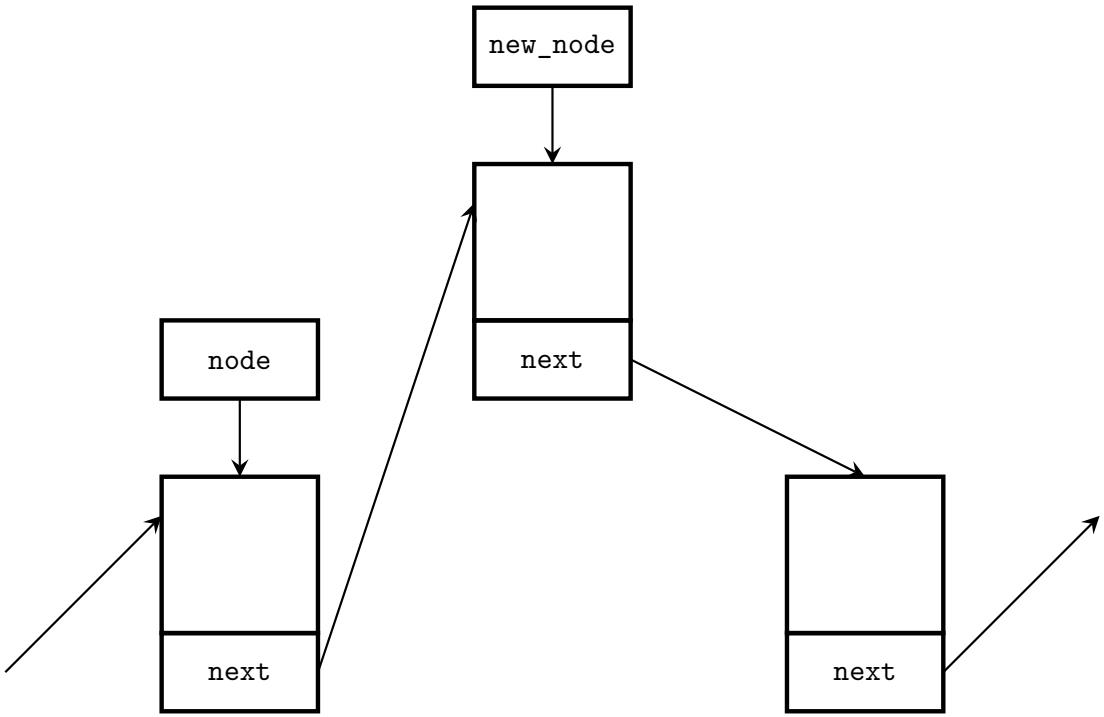
Funkcja `add_in_middle_or_at_back()`



Sytuacja po wykonaniu wiersza nr 4 funkcji `add_in_middle_or_at_back()`

Wstawianie elementu wewnątrz i na końcu listy

Funkcja `add_in_middle_or_at_back()`



Sytuacja po wykonaniu wiersza nr 5 funkcji `add_in_middle_or_at_back()`

Wstawianie elementu wewnątrz i na końcu listy

Funkcja `add_in_middle_or_at_back()`

Zastanówmy się, jak zadziała funkcja `add_in_middle_or_at_back()`, jeśli przez parametr `node` zostanie jej przekazany adres na ostatni element listy. W takiej sytuacji, w wierszu nr 4, polu `next` nowego elementu zostanie nadana wartość `NULL`, bowiem taką właśnie wartość ma pole `next` ostatniego elementu listy, wskazywanego przez `node`. W wierszu nr 5, w polu `next` elementu wskazywanego przez `node` zostanie zapisany adres dodawanego elementu listy. Tym samym stanie się on nowym ostatnim elementem listy i będzie spełniał najważniejszy warunek stawiany takiemu elementowi - jego pole `next` będzie miało wartość `NULL`.

Wstawianie elementu do listy - podsumowanie

Funkcja `add_node()` mogłaby być zapisana bez podziału na funkcje pomocnicze, ale wówczas jej zapis byłby dłuższy i prawdopodobnie mniej czytelny. Jeśli chcielibyśmy zwiększyć efektywność tej funkcji, to moglibyśmy wszystkie funkcje pomocnicze zadeklarować jako funkcje `static inline`, wówczas mogłyby być one potraktowane przez kompilator podobnie jak makra przez preprocesor. Opisany sposób implementacji operacji wstawiania nowego elementu do listy nie jest jedynym słusznym rozwiązaniem. Można funkcję `add_node()` zapisać w innej postaci, np. takiej do której przekazywany jest jako argument wywołania adres wskaźnika na listę.

Operacja usuwania elementu z listy

Operacja usuwania pojedynczego elementu z jednokierunkowej listy liniowej, podobnie jak operacja dodawania musi być przeprowadzana na liście zawierającej co najmniej jeden element. Element zostanie usunięty, jeśli będzie zawierał wskazaną liczbę. Jeśli na liście będzie więcej elementów spełniających to kryterium, to zostanie usunięty pierwszy taki napotkany element. Po wykonaniu tej operacji lista powinna być pomniejszona o jeden element, ale spójna lub powinna być pusta.

Operacja usuwania elementu z listy

Implementacja

Definiując funkcję implementującą usuwanie pojedynczego elementu z listy należy rozważyć i odpowiednio oprogramować cztery następujące przypadki:

1. usuwany element jest pierwszym elementem na liście,
2. element usuwany znajduje się wewnątrz listy,
3. usuwany element jest ostatnim elementem na liście,
4. na liście nie ma elementu, który należałoby usunąć; lista powinna pozostać niezmieniona.

Podobnie jak wstawianiem, usuwaniem elementu będzie zajmowała się w programie pojedyncza funkcja, która będzie korzystała z kilku funkcji pomocniczych. **Wszystkie te funkcje zostaną opisane w odwrotnym porządku, niż są zdefiniowane w programie.**

Operacja usuwania elementu z listy

Funkcja delete_node()

```
1  struct list_node *delete_node(struct list_node *list_pointer,
2                                                    int data)
3  {
4      if(list_pointer) {
5          if(list_pointer->data==data)
6              return delete_at_front(list_pointer);
7          else {
8              struct list_node *previous =
9                              find_previous_node(list_pointer,data);
10             delete_middle_or_last_node(previous);
11         }
12     }
13     return list_pointer;
14 }
```

Operacja usuwania elementu z listy

Funkcja `delete_node()`

Za usunięcie pojedynczego elementu z listy odpowiedzialna jest funkcja `delete_node()`. Przyjmuje ona jako argumenty wywołania wskaźnik na listę (jej pierwszy element) i liczbę, którą ma zawierać usuwany element. Wartością zwracaną przez tę funkcję jest adres pierwszego elementu, który w miejscu jej wywołania powinien być zapisany do wskaźnika na listę. Jeśli usuwany będzie pierwszy element listy, to funkcja zwróci adres elementu, który zastąpi go w tej roli, w pozostałych przypadkach zostanie zwrócony przez nią adres przekazany jej przez pierwszy parametr. W wierszu nr 4 funkcja sprawdza, czy lista, z której ma być usunięty element istnieje. Jeśli tak, to przystępuje do zlokalizowania elementu i, jeśli on istnieje, do jego usunięcia. W wierszu nr 5 funkcja sprawdza, czy elementem usuwanym nie powinien być bieżący pierwszy element listy. W tym celu porównuje przekazaną jej przez drugi parametr liczbę z wartością pola `data` tego elementu.

Operacja usuwania elementu z listy

Funkcja `delete_node()`

Jeśli ten warunek okaże się prawdziwy, to `delete_node()` wywołuje funkcję `delete_at_front()`, która dokonuje właściwego usunięcia i zwraca adres nowego początku listy, który jest następnie zwracany przez opisywaną funkcję i po tym kończy się jej działanie. W przeciwnym przypadku funkcja `delete_node()` musi znaleźć element, który powinien być usunięty z listy. W tym celu wywołuje ona funkcję `find_previous_node()`, która zgodnie z nazwą zwraca adres elementu, który **poprzedza** element do usunięcia. Może się jednak tak zdarzyć, że element ten nie będzie istniał. Wówczas funkcja `find_previous_node()` zwróci adres ostatniego elementu listy. Wszystkie trzy pozostałe przypadki usunięcia elementu z listy są obsługiwane przez wywołanie funkcji `delete_middle_or_last_node()`, która będzie opisana na kolejnych slajdach.

Usunięcie na początku listy

Funkcja `delete_at_front()`

```
1  struct list_node *delete_at_front(struct list_node *list_pointer)
2  {
3      struct list_node *next = list_pointer->next;
4      free(list_pointer);
5      return next;
6  }
```

Usunięcie na początku listy

Funkcja `delete_at_front()`

Usunięciem elementu znajdującego się na początku listy zajmuje się funkcja `delete_at_front()`. Przekazywany jest do niej adres aktualnego pierwszego elementu listy, a zwraca ona adres elementu, który przed jej wykonaniem jest drugi na liście, a po jej wykonaniu będzie pierwszy. Wynik jej działania będzie zwrócony przez funkcję `delete_node()` i ostatecznie zapisany we wskaźniku listy. Działanie opisywanej funkcji jest zbliżone do działania funkcji `pop()` znanej z wykładu o stosie. Ponieważ jest ona wywoływana z poziomu `delete_node()` to nie jest konieczne sprawdzanie, czy przekazany do niej wskaźnik na listę jest poprawny. W wierszu nr 3 funkcja zapamiętuje we wskaźniku lokalnym `next` adres drugiego elementu na liście, następnie zwalnia pamięć przydzieloną na pierwszy element (wiersz nr 4) i kończy swoje działanie zwracając adres nowego pierwszego elementu listy (wiersz nr 5).

Usunięcie wewnątrz i na końcu listy

Funkcja `find_previous_node()`

```
1  struct list_node *find_previous_node
2                          (struct list_node *list_pointer, int data)
3  {
4      struct list_node *previous = NULL;
5      while(list_pointer && list_pointer->data!=data) {
6          previous=list_pointer;
7          list_pointer=list_pointer->next;
8      }
9      return previous;
10 }
```

Usunięcie wewnątrz i na końcu listy

Funkcja `find_previous_node()`

Funkcja `find_previous_node()` odpowiedzialna jest za znalezienie na liście elementu, poprzedzającego ten, który należy usunąć. Jeśli element do usunięcia nie istnieje, to zwróci ona adres ostatniego elementu na liście. Proszę zwrócić uwagę na podobieństwo definicji tej funkcji do `find_spot()`. Odnalezienie elementu w liście wykonywane jest wewnątrz pętli `while`. Do „poruszania" się po liście wykorzystywane są dwa wskaźniki: `list_pointer` i `previous`, który pełni taką samą rolę, jak jego odpowiednik z funkcji `find_spot()`. Pętla z wiersza nr 5 funkcji kończy się, jeśli wskaźnik na listę będzie miał wartość `NULL`, lub jeśli będzie wskazywał element, którego pole `data` będzie miało taką samą wartość, jak liczba przekazana do funkcji przez drugi parametr. Będzie to element do usunięcia, ale funkcja zwróci adres elementu go **poprzedzającego**. Warto zwrócić uwagę, że wyrażenia w warunku kontynuacji pętli `while` nie mogą być zamienione miejscami. Gdyby tak było to kod sięgałby do pola `data` elementu listy przez sprawdzeniem, czy on faktycznie istnieje. To mogłoby spowodować awarię programu.

Usunięcie wewnątrz i na końcu listy

Funkcja `delete_middle_or_last_node()`

```
1  void delete_middle_or_last_node(struct list_node *previous)
2  {
3      struct list_node *node = previous->next;
4      if(node) {
5          previous->next = node->next;
6          free(node);
7      }
8  }
```

Usunięcie wewnątrz i na końcu listy

Funkcja `delete_middle_or_last_node()`

Funkcja `delete_middle_or_last_node()` wbrew swojej nazwie obsługuje także ostatni z opisanych wcześniej przypadków - kiedy element do usunięcia nie istnieje, to nie podejmuje żadnych czynności na liście i pozostawia ją w takim stanie, w jakim ją zastała. Do tej funkcji przekazywany jest wskaźnik na element, który może poprzedzać element do usunięcia. Podobnie, jak w przypadku wcześniej opisywanych funkcji pomocniczych, nie trzeba sprawdzać, czy ten wskaźnik jest poprawny, gdyż funkcja `find_previous_node()` zawsze zwraca wskaźnik o wartości różnej od `NULL`. W wierszu trzecim funkcja zapamiętuje wartość pola `next` elementu wskazywanego przez parametr `previous` w lokalnym wskaźniku `node`. Jeśli ten wskaźnik nie jest pusty, co jest sprawdzane w wierszu nr 4, to znaczy, że istnieje element, który należy usunąć z listy. Może to być ostatni element na liście lub znajdujący się w jej wnętrzu. Okazuje się, że oba przypadki mogą być obsłużone za pomocą tego samego kodu.

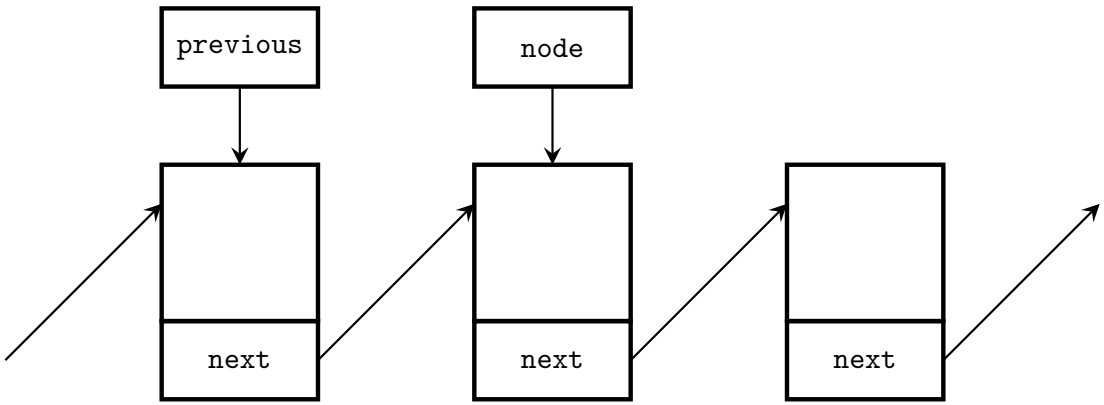
Usunięcie wewnątrz i na końcu listy

Funkcja `delete_middle_or_last_node()`

W wierszu nr 5 do pola `next` elementu wskazywanego przez wskaźnik `previous` zapisywany jest adres znajdujący się w polu `next` elementu wskazywanego przez wskaźnik `node`. Jeśli to pole zawierało pusty wskaźnik, to znaczy, że usuwany jest ostatni element z listy i teraz element wskazywany przez `previous` nim zostanie. Dzięki instrukcji przypisania z piątego wiersza w jego polu `next` znajdzie się wartość `NULL`, czyli taka jaką powinno cechować się pole ostatniego elementu na liście. Jeśli jednak element wskazywany przez `node` nie jest ostatni na liście, to wykonanie instrukcji z wiersza nr 5 spowoduje jego odłączenie od reszty listy. W wierszu nr 6 zwalniana jest pamięć przydzielona na element wskazywany przez `node` i kończone jest działanie funkcji. Kolejne slajdy ilustrują działanie opisywanej funkcji dla przypadku, kiedy należy usunąć element z wnętrza listy.

Usunięcie wewnątrz i na końcu listy

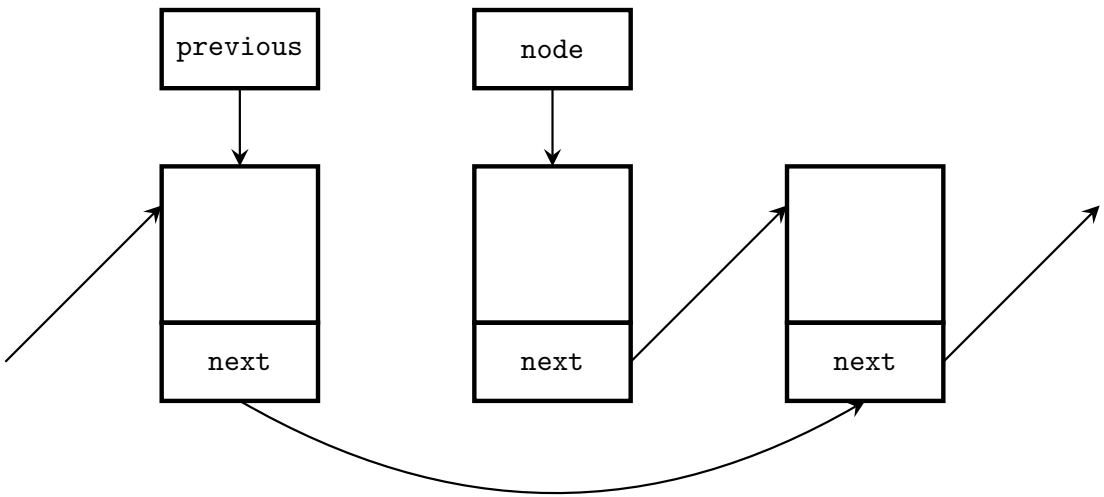
Funkcja `delete_middle_or_last_node()`



Sytuacja przed wykonaniem wiersza nr 5 funkcji `delete_middle_or_last_node()`

Usunięcie wewnątrz i na końcu listy

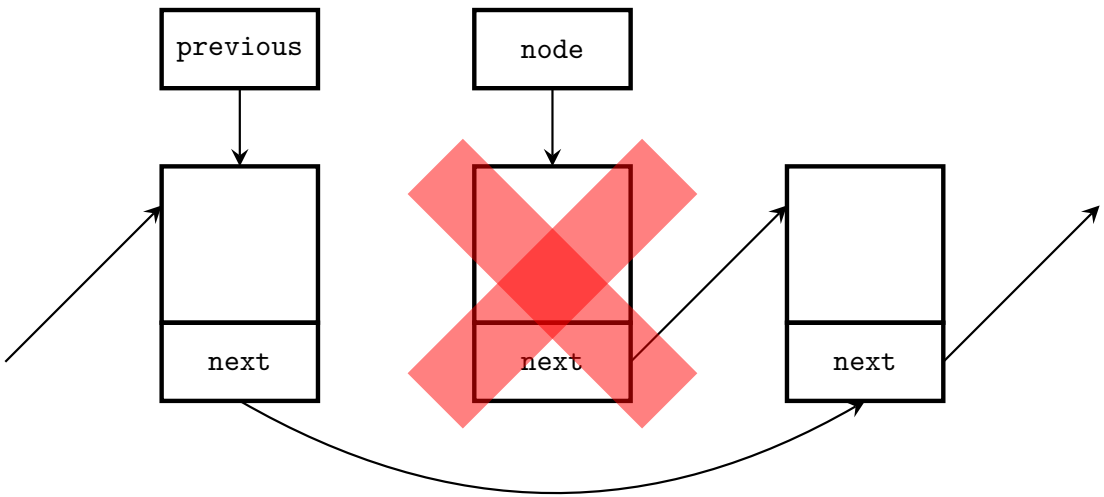
Funkcja `delete_middle_or_last_node()`



Sytuacja po wykonaniu wiersza nr 5 funkcji `delete_middle_or_last_node()`

Usunięcie wewnątrz i na końcu listy

Funkcja `delete_middle_or_last_node()`



Sytuacja po wykonaniu wiersza nr 6 funkcji `delete_middle_or_last_node()`

Usuwanie elementu z listy - podsumowanie

Podobnie jak `add_node()` funkcja `delete_node()` mogłaby być napisana bez podziału na funkcje pomocnicze, ale również w tym przypadku jej kod byłby mniej czytelny. Można podnieść jej efektywność definiując wszystkie funkcje pomocnicze jako `static inline`. Również ta funkcja może zostać zaimplementowana w inny sposób, niż ten, który został przedstawiony na tym wykładzie.

Operacja wyświetlenia zawartości listy

Wyświetlenie zawartości wszystkich elementów listy jest zrealizowane analogicznie do wyświetlenia zwartości kolejki FIFO, dlatego funkcja `print_list()` nie będzie dokładniej tutaj opisywana.

Operacja wyświetlenia zawartości listy

Funkcja `print_list()`

```
1  void print_list(struct list_node *list_pointer)
2  {
3      while(list_pointer) {
4          printf("%d ",list_pointer->data);
5          list_pointer=list_pointer->next;
6      }
7      puts("");
8  }
```

Operacja usuwania listy

Usunięcie listy polega na usunięciu wszystkich jej elementów, czyli zwolnieniu pamięci przeznaczonej na poszczególne z nich, począwszy od pierwszego elementu. Po jej wykonaniu wskaźnik na listę powinien mieć wartość `NULL`. Jest ona zaimplementowana w funkcji `remove_list()`.

Operacja usuwania listy

Funkcja `remove_list()`

```
1  void remove_list(struct list_node **list_pointer)
2  {
3      while(*list_pointer) {
4          struct list_node *next = (*list_pointer)->next;
5          free(*list_pointer);
6          *list_pointer = next;
7      }
8  }
```

Operacja usuwania listy

Funkcja `remove_list()`

Do funkcji `remove_list()` przekazywany jest wskaźnik na wskaźnik listy. W pętli `while` usuwane są kolejne elementy listy do momentu aż ta lista stanie się pusta, a wskaźnik na nią będzie miał wartość `NULL`. Proszę zwrócić uwagę, że wiersze nr 4, 5 i 6 tej funkcji są bardzo podobne do instrukcji umieszczonych w funkcji `pop()` znanej z wykładu o stosie. Elementy listy są niszczone począwszy od pierwszego, a skończywszy na ostatnim. Proszę zwrócić także uwagę na instrukcję przypisania z wiersza nr 4. Nawiasy okrągłe po prawej stronie tej instrukcji są konieczne, gdyż określają kolejność wykonania operatorów - dereferencja wskaźnika musi nastąpić przed sięgnięciem do pola `next` wskazywanego przez niego elementu. Bez nawiasów te operatory byłby wykonywane w złej kolejności, co byłoby zgłoszone jako błąd przez kompilator.

Funkcja `main()`

W funkcji `main()` zostaną wywołane wszystkie opisane funkcje, które realizują podstawowe operacje na liście. Aby przetestować ich działanie należy wywołać je z takimi wartościami liczbowymi, aby zostały wykonane wszystkie przypadki w nich oprogramowane, do których zaliczają się:

- dodanie elementu na początku listy,
- dodanie elementu wewnątrz listy,
- dodanie elementu na końcu listy,
- usunięcie elementu z początku listy,
- usuniecie elementu z wnętrza listy,
- usunięcie elementu z końca listy,
- próba usunięcia 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          list_pointer=add_node(list_pointer,i);
7      for(i=6; i<10; i++)
8          list_pointer=add_node(list_pointer,i);
9      print_list(list_pointer);
```

Funkcja `main()`

Część pierwsza

W pierwszej części funkcji `main()` tworzona jest lista składająca się z pierwszego elementu o wartości 1, a następnie dodawane są do niej elementu o wartościach od 2 do 4 i od 6 do 9. Po wykonaniu tych czynności zawartość listy jest wypisywana na ekranie.

Funkcja `main()`

Część druga

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

Funkcja `main()`

Część druga

W drugiej części funkcji `main()` do listy są dodawane elementy kolejno o wartościach: `0` (będzie umieszczony na początku listy), `5` (będzie umieszczony wewnątrz listy), `7` (będzie umieszczony wewnątrz listy, przed elementem o tej samej wartości), `10` (będzie umieszczony na końcu listy). Po każdej takiej operacji wypisywana jest zawartość tej listy na ekranie, aby użytkownik mógł sprawdzić, czy ta operacja przebiegła prawidłowo.

Funkcja main()

Część trzecia

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

Funkcja `main()`

Część trzecia

W trzeciej części funkcji `main()` usuwane są z listy kolejno elementy o wartościach: `0` (jest na początku listy), `1` (jest na nowym początku listy), `1` (po wcześniejszym usunięciu nie ma go już na liście), `5` (jest wewnątrz listy), `10` (jest na końcu listy). Po wykonaniu każdej takiej operacji zawartość listy jest wypisywana na ekranie. Na koniec lista jest usuwana za pomocą funkcji `remove_list()` i kończone jest działanie funkcji `main()`.

Zastosowania

Zaimplementowane w postaci dynamicznej listy liniowe, zarówno jednokierunkowe, jak i dwukierunkowe, które będą omawiane na kolejnych wykładach, mają liczne zastosowania. Do typowych przykładów należą systemy operacyjne, gdzie są wykorzystywane dosyć powszechnie. W jądrze systemu Linux ich używanie jest na tyle częste, że programiści stworzyli osobną implementację tej listy, z której mogą korzystać w wielu miejscach systemu. Większość współczesnych języków programowania posiada również gotowe implementacje takich list, dostarczane w standardowych bibliotekach języka (np. Java) lub dostępne jako jego integralny element (np. Python).

Podsumowanie

Jednokierunkowe listy liniowe mogą być zaimplementowane z użyciem wartowników, podobnie jak kolejki. W takim przypadku, na początku działania programu tworzony jest jeden lub dwa elementy pełniące rolę atrap. Dzięki nim eliminowana jest konieczność sprawdzania niektórych warunków w implementacjach operacji dodawania i usuwania elementów z listy. Możliwa jest również realizacja opisywanych list z użyciem tablic wielowymiarowych. Pierwszy wiersz takiej tablicy zawiera wartości elementów listy, a drugi wartości indeksów pełniące rolę adresów następnych elementów na liście. Niestety pojemność takiej listy jest ograniczona, a operacje wstawiania są dosyć kłopotliwe, bo wymagają kopiowania części elementów tablicy. Takiej czynności może wymagać również usuwanie elementów z listy. Można je jednak zrealizować inaczej, wymaga to jednak przyjęcia, że określona wartość elementu tablicy w drugim jej wierszu będzie oznaczała elementu usunięty. Takie elementy będzie można wykorzystać później przy wstawianiu nowych elementów do listy. Musi również istnieć wartość indeksu, która będzie odpowiednikiem wartości NULL.

Podsumowanie

Do realizacji jednokierunkowej listy liniowej można użyć także tablicy jednowymiarowej. Co drugi element w takiej liście będzie pełnił rolę wskaźnika `next`. Obie implementacje się komplikują, jeśli na liście muszą być przechowywane wartości innych typów, niż proste typy danych. Jednakże taka forma realizacji listy może się okazać konieczna w przypadku starszych języków programowania lub wtedy, gdy program jest tworzony dla systemu komputerowego o ograniczonych zasobach pamięciowych.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!