Podstawy Programowania 2

Dwukierunkowa lista liniowa

Arkadiusz Chrobot

Zakład Informatyki

6 kwietnia 2020

Plan

Wstęp

Dwukierunkowa lista liniowa (ang. *doubly linked list*) jest kolejnym po liście jednokierunkowej przykładem listy liniowej. Te dwie struktury są do siebie dosyć zbliżone. Od strony użytkowej różnie je to, że dwukierunkowa lista liniowa posiada budowę, która umożliwia w prosty sposób przeglądanie jej, zgodnie z jej nazwą, w dwóch kierunkach. Na tym wykładzie zostanie przedstawiona implementacja tej listy w postaci dynamicznej struktury danych.

Implementacja

Podobnie jak w przypadku poprzednio opisywanej listy, dwukierunkowa lista liniowa zostanie przedstawiona na przykładzie programu, który korzysta z niej do przechowywania liczb naturalnych uszeregowanych nierosnąco. Będzie to zatem lista uporządkowana. Podobnie jak w przypadku jednokierunkowej listy liniowej najpierw zdefiniujemy typ bazowy listy i jej wskaźnik, a następnie pięć podstawowych operacji: tworzenia listy, dodawania nowego elementu do listy, usuwania elementu z listy, wyświetlania zawartości listy na ekran i usuwania listy.

Typ bazowy i wskaźnik listy

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

Typ bazowy

Zaprezentowany na poprzednim slajdzie typ bazowy dwukierunkowej listy liniowej różni się od typu bazowego innych poznanych przez nas dotychczas struktur danych posiadaniem dodatkowego pola wskaźnikowego. To pole, nazwane w przykładowym typie bazowym `previous`, służy do przechowywania adresu elementu listy, który poprzedza dany element. W dwukierunkowej liście liniowej może istnieć tylko jeden element, który będzie miał zapisaną wartość `NULL` w tym polu i jest to pierwszy element tej listy. Istnieje odmiana listy dwukierunkowej, której typ bazowy jest taki sam jak dla listy jednokierunkowej. Nie posiada ona nazwy w języku polskim, w języku angielskim nazywa się ją XOR *linked list*. Dwa adresy są umieszczane w jednym polu wskaźnikowym poprzez wykonanie na nich operacji bitowej różnicy symetrycznej. Oszczędność pamięci jest w przypadku tej listy uzyskiwana kosztem bardziej skomplikowanych operacji. Taka implementacja nie będzie tutaj opisywana. Zaprezentowany przykładowy typ bazowy można dowolnie rozbudowywać i zmieniać, o ile zostaną zachowane oba pola wskaźnikowe niezbędne do budowy struktury dwukierunkowej listy.

Wskaźnik listy

Wraz z definicją typu bazowego została wykonana deklaracja wskaźnika listy (zmienna `list_pointer`). W przeciwieństwie do listy jednokierunkowej nie musi on wskazywać początkowego elementu listy, wystarczy, aby wskazywał *dowolny* jej element. Niemniej jednak wygodniej będzie, jeśli każda operacja na liście po zakończeniu zostawi ten wskaźnik na pierwszym elemencie. Takie podejście zastosujemy w prezentowanym programie.

Na listingu z definicją typu bazowego i deklaracją wskaźnika listy umieszczone są także instrukcje dołączające pliki nagłówkowe zawierające definicje funkcji używanych w programie.

Tworzenie listy

Podobnie jak w przypadku jednokierunkowej listy liniowej operacja tworzenia listy, która równoważna jest utworzeniu jej pierwszego elementu i zapamiętaniu jego adresu we wskaźniku listy, będzie zrealizowana w osobnej funkcji o nazwie `create_list()`. Jej kod źródłowy jest zamieszczony na następnym slajdzie.

Tworzenie listy

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

Tworzenie listy

Ponieważ definicja funkcji `create_list()` w wersji dla dwukierunkowej listy liniowej niewiele różni się od jej odpowiedniczki dla listy jednokierunkowej, to tutaj zostaną opisane jedynie najważniejsze różnice, które w całości zawierają się w wierszu nr 7. Otóż w nim wartość `NULL` jest przypisywana nie tylko do wskaźnika `next`, ale również do wskaźnika `previous` pierwszego elementu. W przypadku tworzenia listy będzie on jej pierwszym i zarazem ostatnim elementem. Adres zwrócony przez tę funkcję w miejscu jej wywołania powinien zostać zapisany we wskaźniku listy.

Dodawanie elementu do listy

Podobnie jak w przypadku jednokierunkowej listy liniowej będziemy wymagać, aby operacja dodania nowego elementu do dwukierunkowej listy liniowej była wykonywana na niepustej liście. Przyjmijmy także, że po wykonaniu tej operacji wskaźnik listy powinien wskazywać na jej pierwszy element, a w przypadku, kiedy nie uda się utworzyć nowego elementu, to stan listy powinien pozostać niezmieniony.

Dodawanie elementu do listy

Istnieją trzy przypadki, które należy rozpatrzeć implementując operację dodawania nowego elementu do listy:

1. element jest dodawany na początku listy i zostanie jej pierwszym elementem,
2. element jest dodawany wewnątrz listy,
3. element jest dodawany na końcu listy i zostanie jej ostatnim elementem.

Wszystkie te przy przypadki zostaną oprogramowane w osobnych funkcjach pomocniczych, które będą wywoływane z poziomu pojedynczej funkcji realizującej operację dodania elementu do dwukierunkowej listy liniowej. Najpierw jednak zostaną przedstawione kody źródłowe funkcji pomocniczych.

Dodawanie elementu do listy

Dodanie na początek listy

```
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      list_pointer->previous = new_node;
6      return new_node;
7  }
```

Dodawanie elementu do listy

Dodanie na początek listy

W stosunku do swojej odpowiedniczki dla jednokierunkowej listy liniowej ta funkcja musi uwzględniać istnienie w każdym elemencie drugiego pola wskaźnikowego. Stąd w wierszu nr 5 w polu `previous` bieżącego pierwszego elementu listy zapisywany jest adres nowego elementu. Reszta wykonywanych instrukcji jest taka sama, jak dla jednokierunkowej listy liniowej.

Dodanie elementu do listy

Wyszukanie miejsca

Obsłużenie dwóch pozostałych przypadków będzie wymagało przeszukania listy. W wyniku przeprowadzenia tej operacji oczekujemy, że uzyskamy wskaźnik elementu, **za** którym dodamy nowy. Do jego lokalizacji możemy wykorzystać tę samą funkcję `find_spot()`, którą zdefiniowaliśmy dla jednokierunkowej listy liniowej. Jej kod źródłowy, celem przypomnienia, podany jest na następnym slajdzie. Dokonano w nim tylko jednej zmiany, zmieniono nazwę drugiego parametru.

Dodanie elementu do listy

Wyszukanie miejsca

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

Dodanie elementu do listy

Dodanie wewnątrz listy

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

Dodanie elementu do listy

Dodanie wewnątrz listy

Dodanie elementu wewnątrz dwukierunkowej listy liniowej jest nieco bardziej skomplikowaną operacją niż dodanie elementu do listy jednokierunkowej, z uwagi na konieczność uwzględnienia dodatkowego pola wskaźnikowego. W wierszu nr 4 do pola `previous` nowego elementu jest zapisywany adres elementu, za którym ma on być wstawiony na liście (innymi słowy: ten element będzie go poprzedzał). W polu `next` nowego elementu zapisywany jest z kolei adres elementu, przed którym będzie w liście umieszczony nowy (inaczej: element następujący po nim). Ta operacja wykonywana jest w wierszu nr 5. W wierszu nr 6 do pola `previous` tego elementu jest zapisywany adres nowego. Lewa strona tej instrukcji przypisania ma dosyć skompilowany zapis, ale oznacza on tylko tyle, że za pomocą wskaźnika `next` elementu, który ma poprzedzać w liście nowy funkcja sięga do elementu, który ma się znaleźć za nowym i modyfikuje jego pole `previous`. Dzięki temu nowy element zostaje częściowo włączony do listy.

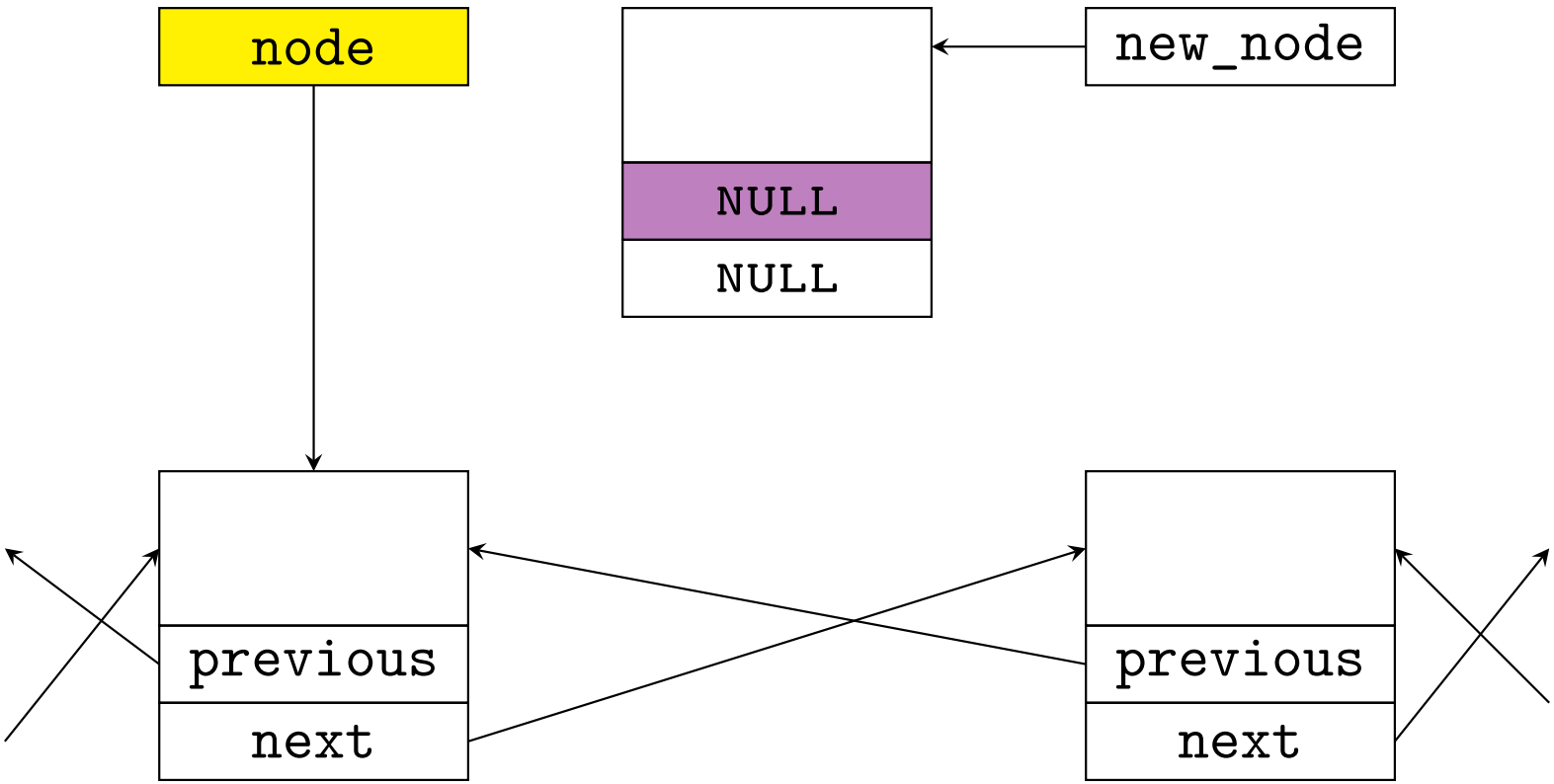
Dodanie elementu do listy

Dodanie wewnątrz listy

Pozostaje tylko w polu `next` elementu poprzedzającego na liście nowy zapisać adres tego nowego elementu (wiersz 7). Proszę zwrócić uwagę, że wiersze 6 i 7 nie mogą być zamienione miejscami. Kolejne slajdy ilustrują działanie funkcji `add_in_middle()`. Kolorem żółtym zaznaczono pola, z których wartości będą kopiowane, a kolorem fioletowym pola, do których wartości będą kopiowane.

Dodanie elementu do listy

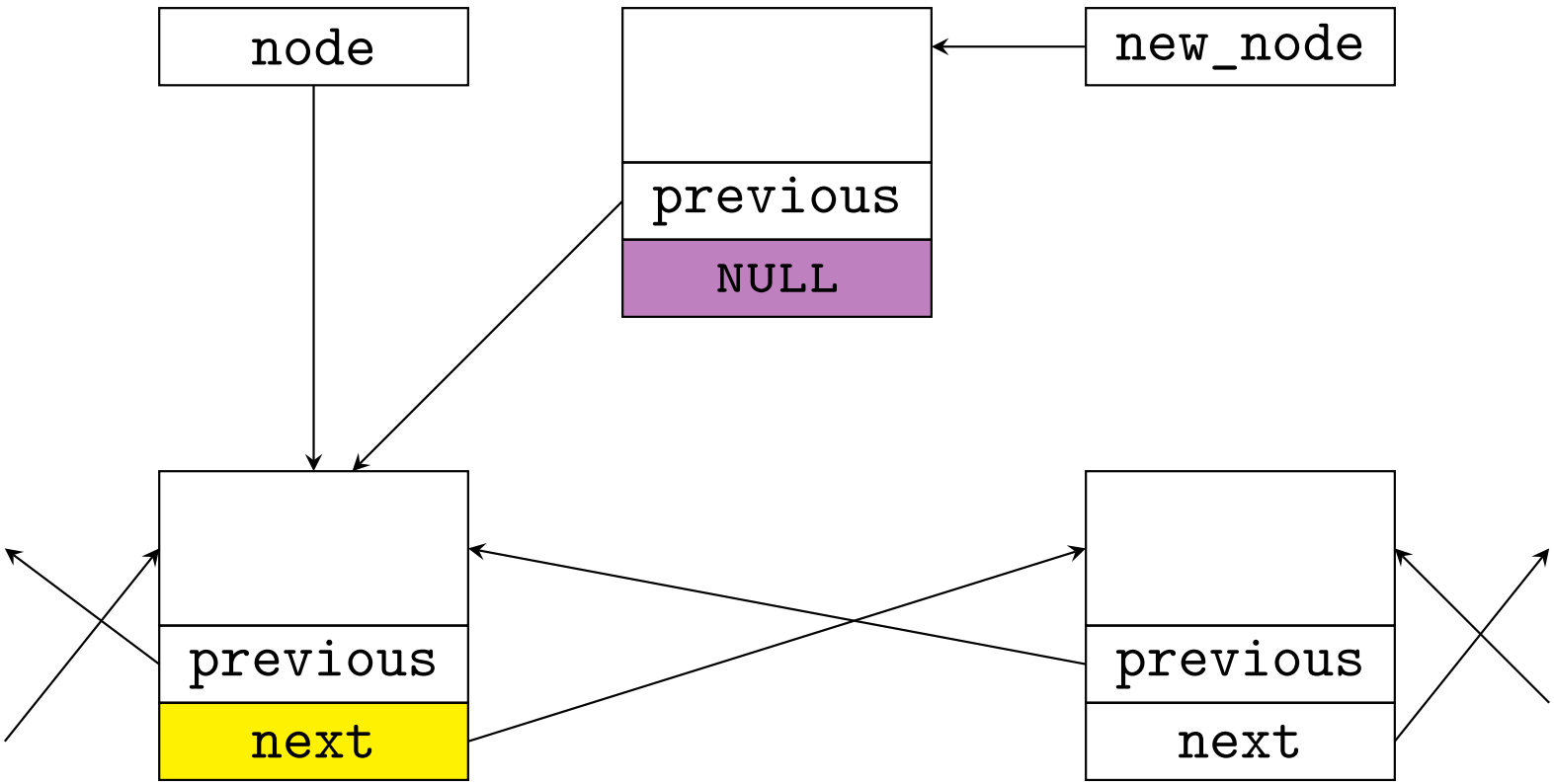
Dodanie wewnątrz listy



Sytuacja przed wykonaniem wiersza nr 4

Dodanie elementu do listy

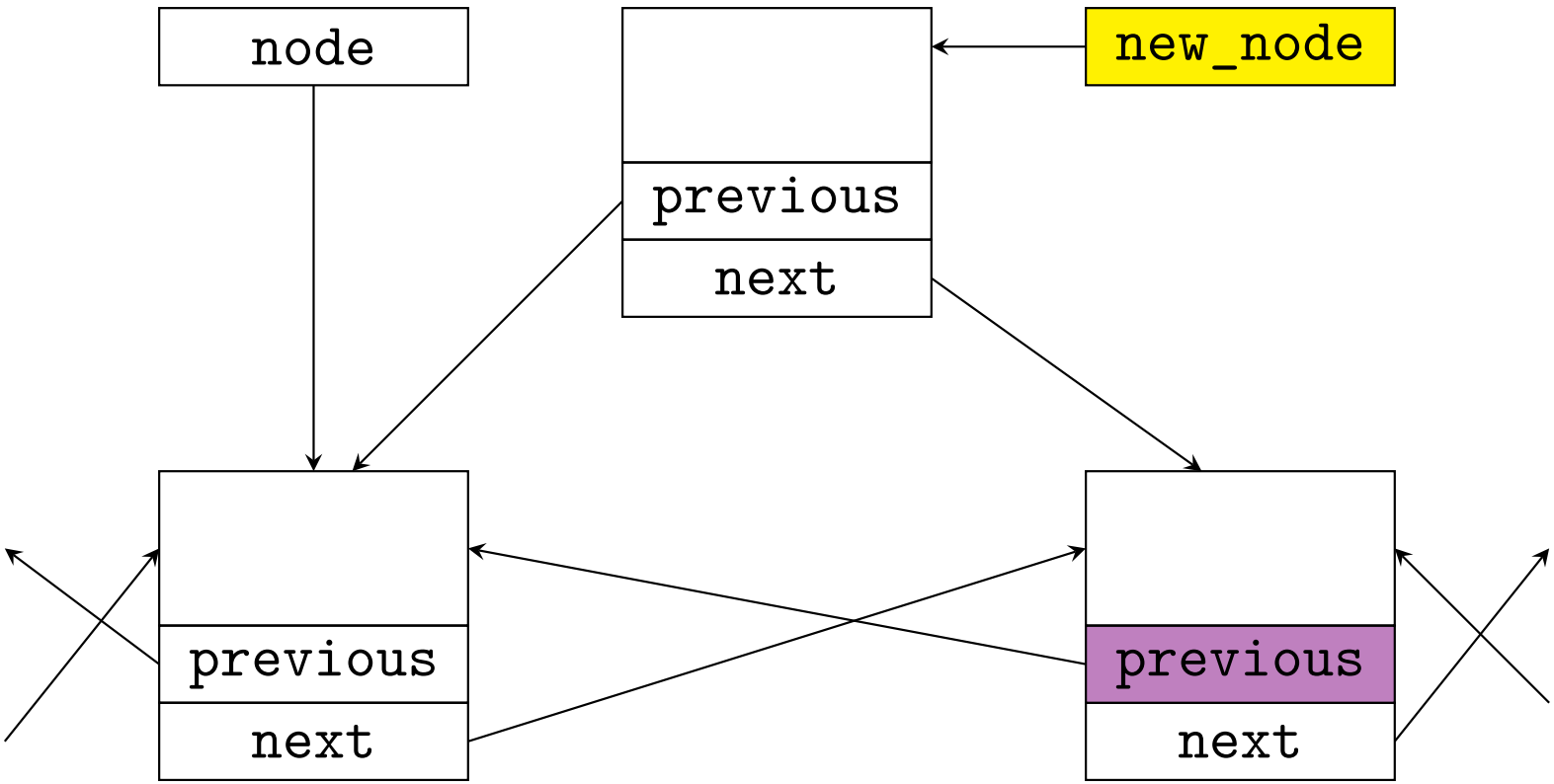
Dodanie wewnątrz listy



Sytuacja po wykonaniu wiersza nr 4

Dodanie elementu do listy

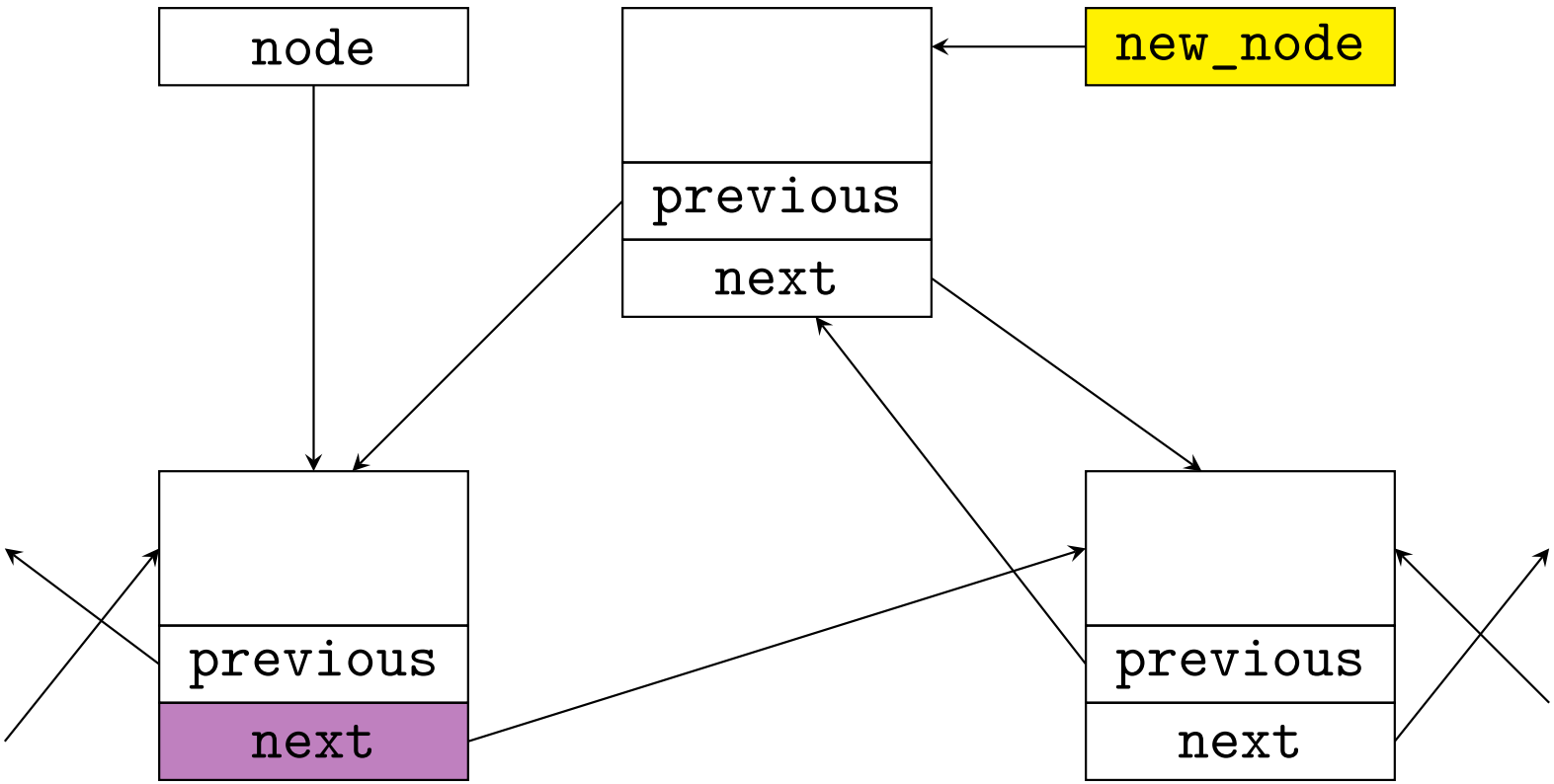
Dodanie wewnątrz listy



Sytuacja po wykonaniu wiersza nr 5

Dodanie elementu do listy

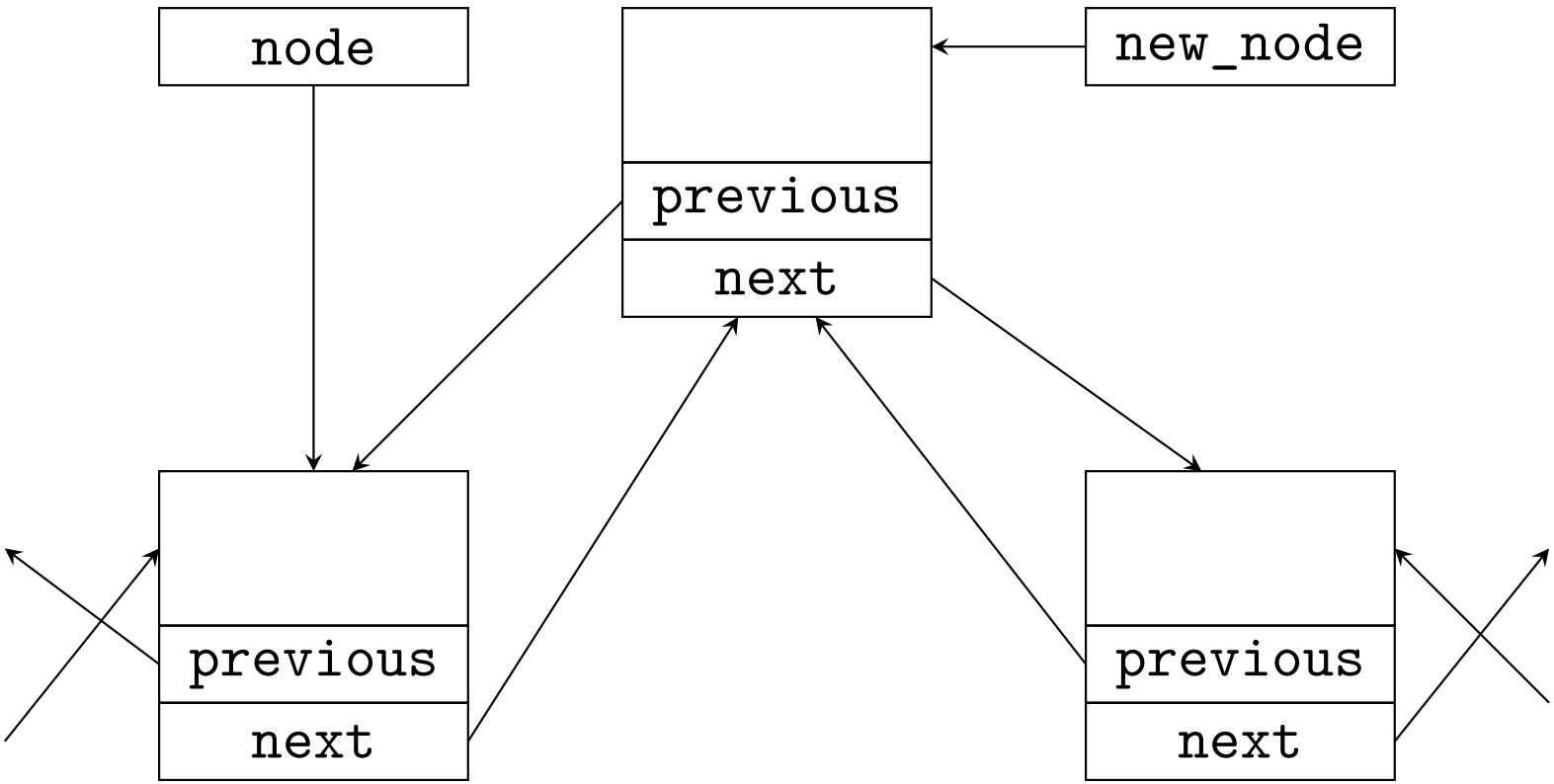
Dodanie wewnątrz listy



Sytuacja po wykonaniu wiersza nr 6

Dodanie elementu do listy

Dodanie wewnątrz listy



Sytuacja po wykonaniu wiersza nr 7

Dodanie elementu do listy

Dodanie na koniec listy

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

Dodanie elementu do listy

Dodanie na koniec listy

Do funkcji `add_at_back()` przekazywane są dwa wskaźniki. Pierwszy z nich wskazuje na bieżący ostatni element listy, a drugi na nowy element listy, który ma być dodany na jej końcu. W wierszu nr 4 w polu `next` aktualnego końcowego elementu listy zapisywany jest adres nowego elementu. Tym samym nowy element zostaje ostatnim elementem listy. Nie jest on jednak w pełni do niej podłączony, dlatego w wierszu nr 5 w polu `previous` tego elementu zapisywany jest adres elementu wskazywanego przez `last_node`. Po wykonaniu tej operacji nowy element staje się integralną częścią listy i jej ostatnim elementem.

Funkcja add_node()

```
1   struct list_node *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               new_node->previous = new_node->next = NULL;
9               if(list_pointer->data>=number)
10                  return add_at_front(list_pointer,new_node);
11              else {
12                  struct list_node *node = find_spot(list_pointer, number);
13                  if(node->next)
14                      add_in_middle(node, new_node);
15                  else
16                      add_at_back(node, new_node);
17              }
18          }
19      }
20      return list_pointer;
21  }
```

Funkcja `add_node()`

Kod funkcji `add_node()` jest nieco bardziej skomplikowany niż jej odpowiedniczki dla jednokierunkowej listy liniowej. Role parametrów, ani znaczenie wartości zwracanej nie uległy zmianie. Funkcja `add_node()` dla dwukierunkowej listy liniowej najpierw sprawdza, czy przekazana jej lista nie jest pusta (wiersz 3). Gdyby tak było, to jej działanie się zakończy wykonaniem wiersza nr 20 - lista pozostanie pusta. W przeciwnym przypadku funkcja przydzieli pamięć na nowy element (wiersze 4 i 5), a następnie sprawdzi, czy ten przydział się powiódł (wiersz nr 6). Jeśli tak będzie, to nastąpi inicjacja pól nowego elementu (wiersze 7 i 8) i rozpoznanie, który z przypadków operacji wstawiania trzeba wykonać. W przeciwnym przypadku działanie funkcji także kończy się wykonaniem wiersza nr 20 i także tym razem stan listy pozostaje niezmieniony. W wierszu nr 9 funkcja sprawdza, czy ma wstawić nowy element na początku listy. Jeśli tak, to wywołuje funkcję pomocniczą `add_at_front()` i po jej wykonaniu kończy swe działanie. W przeciwnym przypadku ustala element listy za którym ma się pojawić nowy przy pomocy wywołania funkcji `find_spot()`.

Funkcja `add_node()`

W wierszu nr 13 sprawdza ona, czy znaleziony element nie jest ostatnim na liście. Jeśli tak, to wywołuje funkcję `add_in_middle()`, a w przeciwnym przypadku `add_at_back()` i kończy swe działanie.

Usuwanie elementu z listy

Operacja usunięcia elementu z dwukierunkowej listy liniowej jest również wykonywana trochę odmiennie niż w przypadku listy jednokierunkowej. Mimo podobieństw różnice są na tyle duże, że jej implementacja jest bardziej skomplikowana. Podobnie jak dla poprzednio przedstawionej listy będziemy wymagać, aby lista po wykonaniu tej operacji przestała istnieć, jeśli była listą jednoelementową, została pomniejszona o jeden element, ale zachowała spójną budowę lub by jej stan nie uległ zmianie, jeśli nie zawiera ona elementu, który należałoby zwolnić.

Usuwanie elementu z listy

Podobnie jak w przypadku listy jednokierunkowej wyróżniamy cztery przypadki, które należy uwzględnić implementując usuwanie elementu z listy:

1. usuwanie pierwszego elementu z listy,
2. usuwanie elementu z wnętrza listy,
3. usuwanie elementu z końca listy,
4. lista nie zawiera elementu, który należy usunąć.

Trzy pierwsze przypadki są obsługiwane przez funkcje pomocnicze, które zostaną przedstawione i opisane na następnych slajdach. Są one wywoływane z poziomu funkcji `delete_node()`, która zostanie przedstawiona i udokumentowana po nich. Czwarty przypadek nie wymaga osobnego podprogramu.

Usuwanie z początku listy

```
1  struct list_node *delete_at_front(struct list_node *list_pointer)
2  {
3      struct list_node *next = list_pointer->next;
4      if(next)
5          next->previous = NULL;
6      free(list_pointer);
7      return next;
8  }
```

Usuwanie z początku listy

Funkcja `delete_at_front()` w porównaniu do swojej odpowiedniczki dla jednokierunkowej listy liniowej musi uwzględniać istnienie pola `previous` w drugim elemencie listy, który znajdzie się na jej początku po usunięciu pierwszego elementu. Zatem w wierszu nr 4 sprawdza ona, czy istnieje kolejny element listy. Jeśli nie, to usuwany element jest ostatnim elementem listy i funkcja przechodzi bezpośrednio do wykonania wiersza nr 6. Jeśli tak, to w polu `previous` kolejnego elementu zapisywana jest wartość `NULL` (wiersz nr 5), bo stanie się on pierwszym elementem listy. Po wykonaniu tej czynności dopiero jest zwalniany dotychczasowy pierwszy element (ponownie wiersz nr 6). Funkcja kończy się zwracając wartość swojego lokalnego wskaźnika o nazwie `next`.

Wyszukiwanie elementu

```
1  struct list_node *find_node(struct list_node *list_pointer,
2                                                  int number)
3  {
4      while(list_pointer&&list_pointer->data!=number)
5          list_pointer = list_pointer->next;
6      return list_pointer;
7  }
```

Wyszukiwanie elementu

Funkcja `find_node()` zwraca adres elementu listy zawierającego w polu `data` liczbę przekazaną jej przez parametr `number`. Za pomocą pierwszego parametru do funkcji jest przekazywany wskaźnik na listę. W pętli `while` (wiersze 4 i 5) funkcja sprawdza każdy element listy, czy zawiera on taką samą liczbę jak parametr `number`. Pętla ta wykonuje się tak długo, aż zostanie znaleziony dany element lub lista się skończy. Wynik tego poszukiwania (adres elementu lub wartość `NULL`) po jej zakończeniu będzie zapisany we wskaźniku `list_pointer` i zawartość tego wskaźnika zostanie przez funkcję zwrócona.

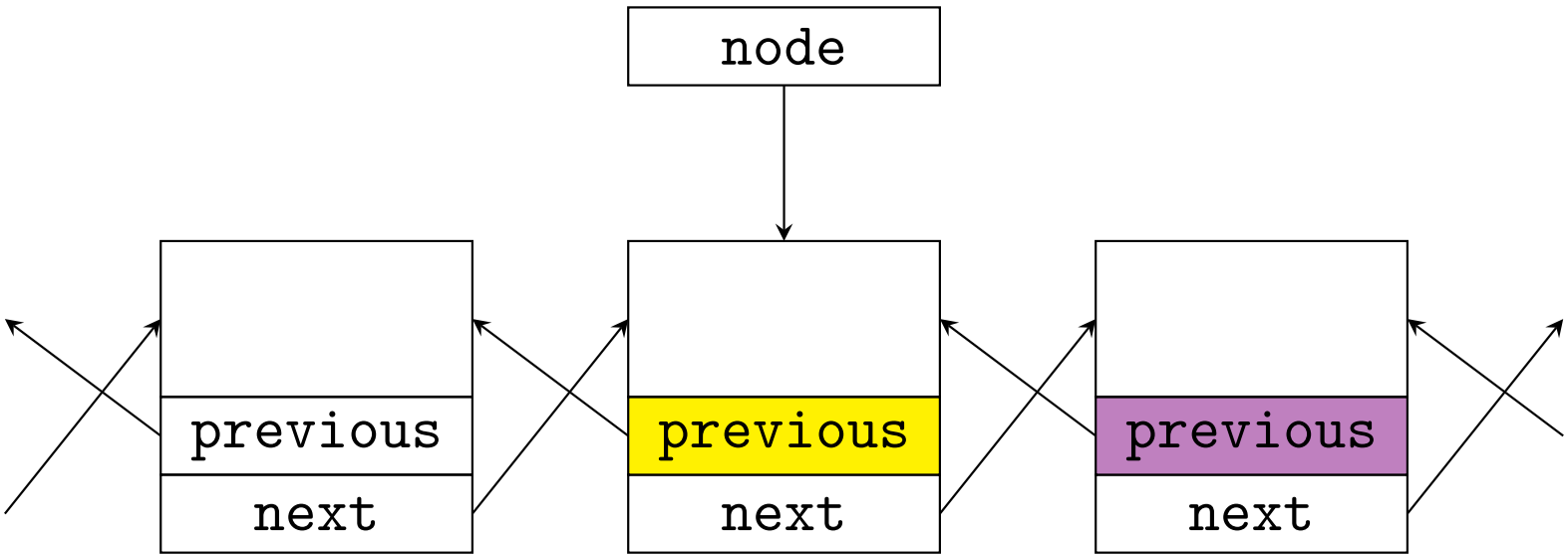
Usuwanie z wnętrza listy

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

Usuwanie z wnętrza listy

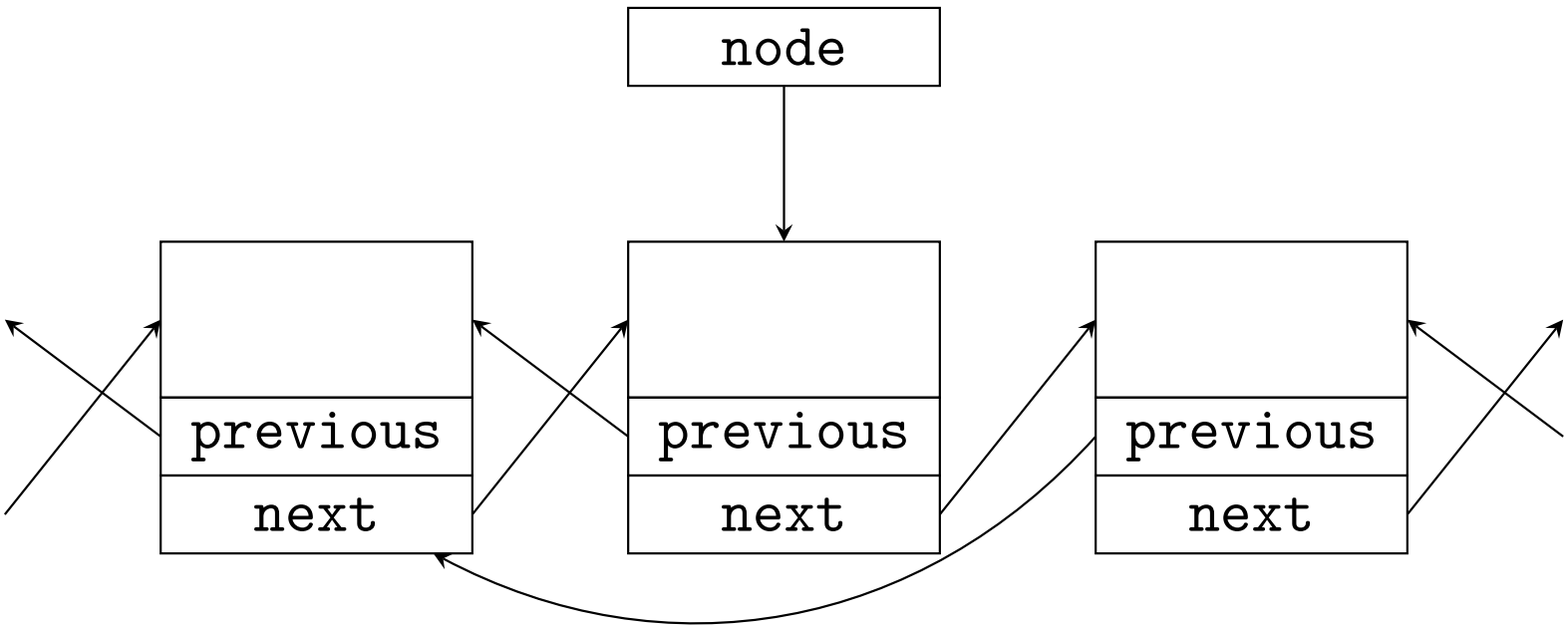
Zaprezentowana na poprzednim slajdzie funkcja usuwa z wnętrza listy element, którego adres został jej przekazany przez parametr. W tym celu najpierw w wierszach 3 i 4 wyłącza go z listy, a następnie zwalnia pamięć na niego przeznaczoną w wierszu nr 5. W wierszu nr 3 funkcja za pomocą przekazanego jej wskaźnika na usuwany element sięga do elementu, który jest na liście następny w stosunku do niego i zapisuje w jego polu `previous` adres elementu poprzedzającego ten wskazywany przez `node`. W wierszu nr 4 wykonywana jest czynność dotycząca przeciwnych kierunków, tzn. za pomocą przekazanego wskaźnika funkcja sięga do elementu poprzedzającego element wskazywany przez `node` (lewa strona instrukcji przypisania) i zapisuje w jego polu `next` adres elementu, który występuje jako następny po tym wskazywanym przez `node` (prawa strona instrukcji przypisania). Kolejne slajdy obrazują działanie tej funkcji. Znaczenie kolorów jest takie samo jak na poprzednio zaprezentowanych ilustracjach.

Usuwanie z wnętrza listy



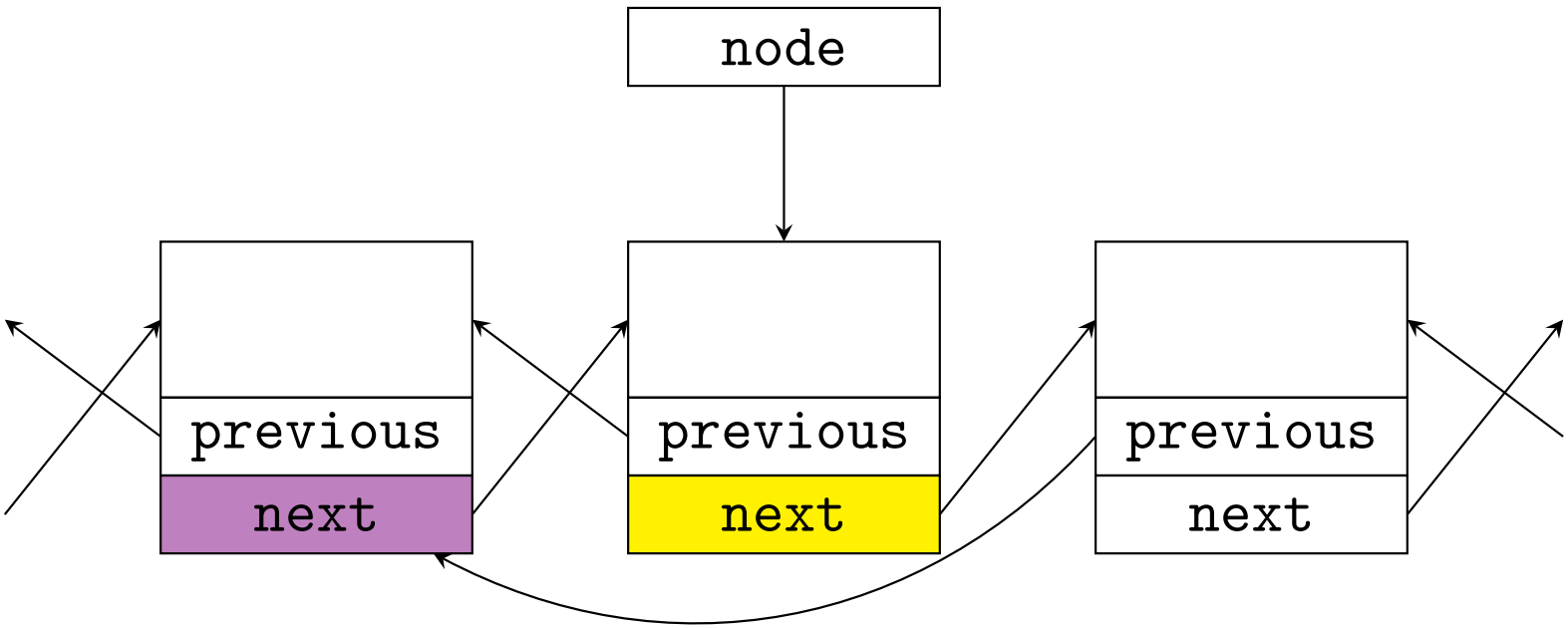
Przed wykonaniem wiersza nr 3

Usuwanie z wnętrza listy



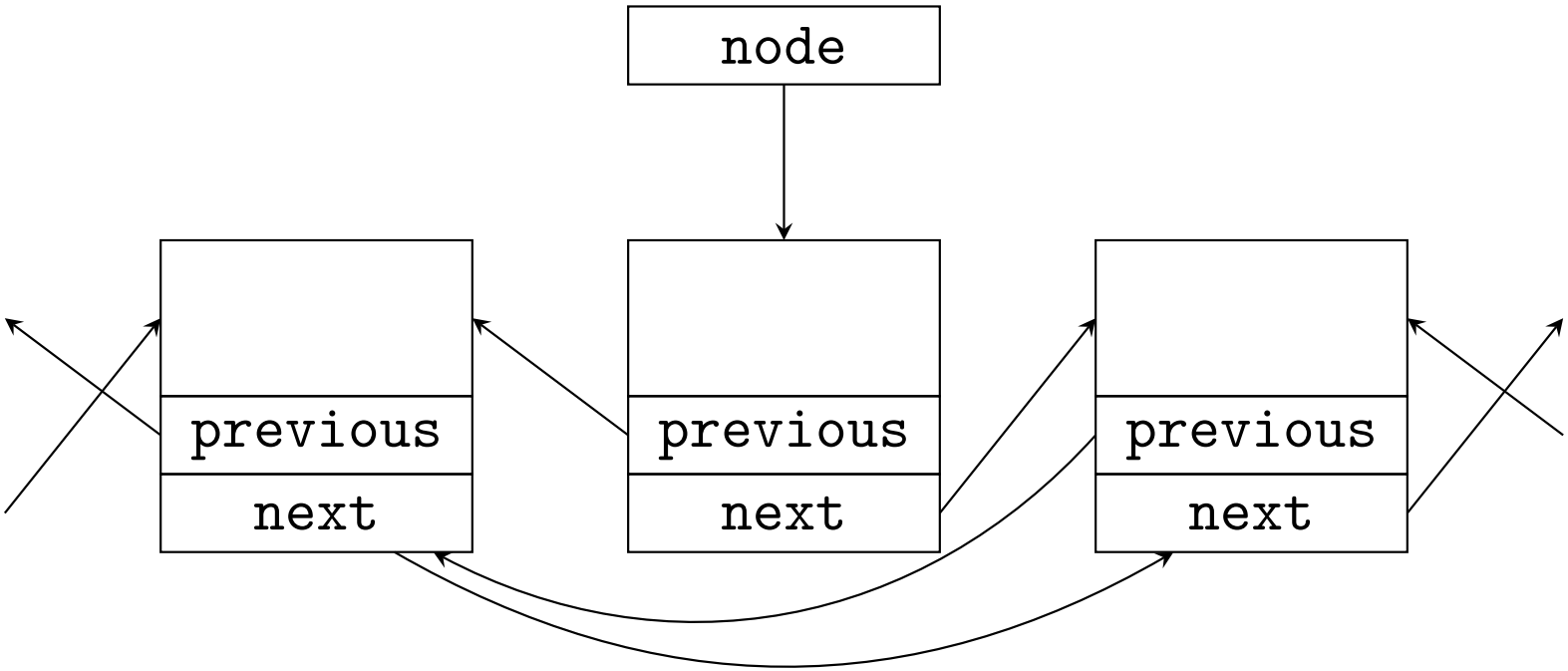
Po wykonaniu wiersza nr 3

Usuwanie z wnętrza listy



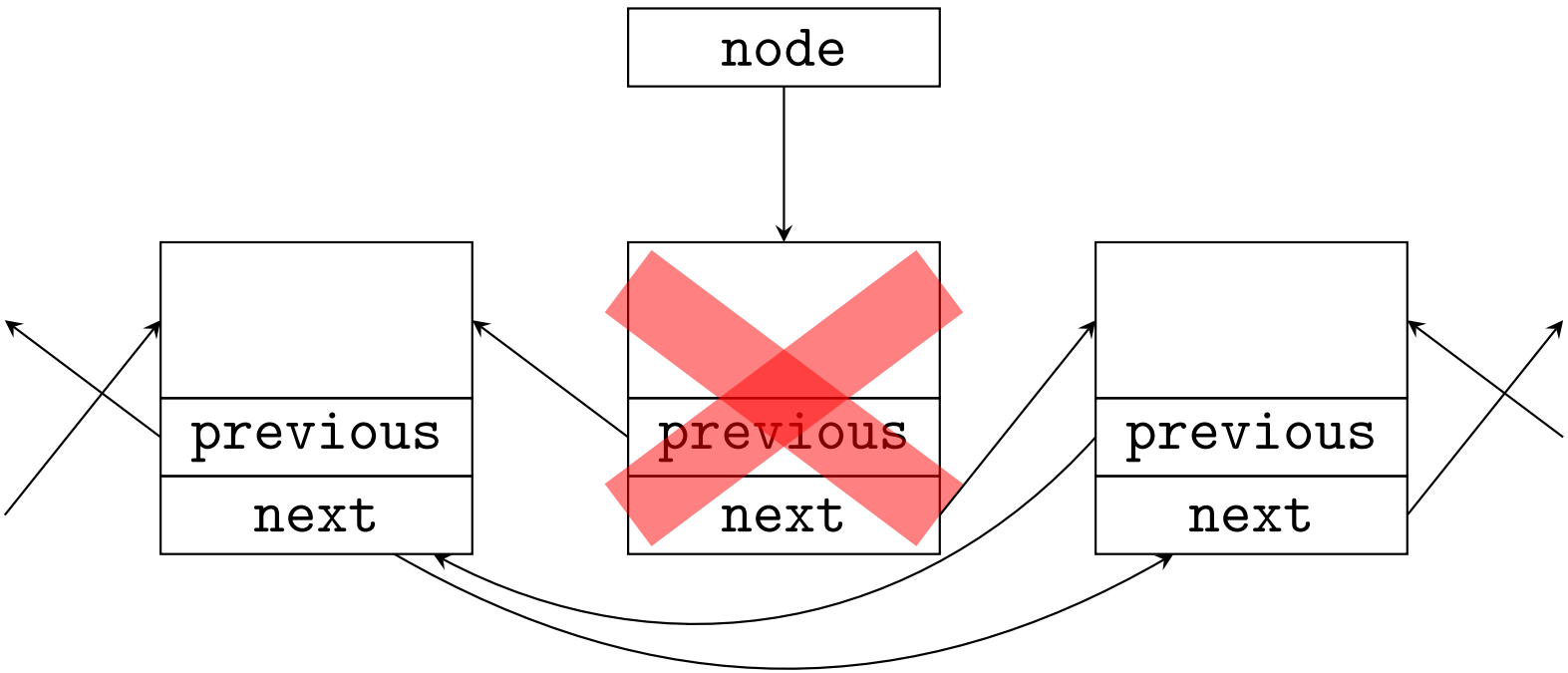
Przed wykonaniem wiersza nr 4

Usuwanie z wnętrza listy



Po wykonaniu wiersza nr 4

Usuwanie z wnętrza listy



Po wykonaniu wiersza nr 5

Usuwanie z końca listy

```
1  void delete_at_back(struct list_node *last_node)
2  {
3      last_node->previous->next = NULL;
4      free(last_node);
5  }
```

Usuwanie z końca listy

Funkcja `delete_at_back()` jest odpowiedzialna za usunięcie ostatniego elementu listy. Wskaźnik na ten element jest jej przekazywany przez parametr. Za jego pomocą w wierszu nr 3 ta funkcja sięga do elementu poprzedzającego element usuwany i w polu `next` tego pierwszego zapisuje wartość `NULL`. Ten element stanie się ostatnim na liście. W wierszu nr 4 funkcja zwalnia pamięć na element wskazywany przez parametr `last_node`.

Usunięcie z listy

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

Usunięcie z listy

Funkcja `delete_node()`, tak jak `add_node()` jest bardziej skomplikowana od swojej odpowiedniczki dla jednokierunkowej listy liniowej. Znaczenie jej parametrów i wartości zwracanej pozostało takie samo. Funkcja na wstępie (wiersz nr 4) sprawdza, czy lista na której ma być wykonana operacja istnieje. Jeśli nie to zwraca wartość wskaźnika `list_pointer`, czyli w tym wypadku `NULL`. Jeśli jednak lista istnieje, to funkcja najpierw sprawdza, czy ma zostać usunięty jej pierwszy element. Jeśli odpowiedź na to pytanie jest prawdziwa, to wywołuje `delete_at_front()`, zwraca jej wynik i kończy swe działanie. W przeciwnym przypadku ustala za pomocą wywołania funkcji `find_node()` adres elementu, który powinien zostać usunięty. Jeśli ta funkcja zwróci wartość `NULL`, co sprawdzane jest w wierszu nr 10, to będzie to oznaczało, że nie ma elementu do usunięcia na liście i `delete_node()` zakończy działanie wykonując wiersz nr 18.

Usunięcie z listy

Jeśli adres zwrócony przez funkcję `find_node()` będzie różny od `NULL`, to funkcja w wierszu nr 11 sprawdzi, czy usuwany element jest wewnątrz listy. Jeśli tak, to wywoła celem jego usunięcia `delete_in_middle()`. Jeśli jednak okaże się, że usunięty powinien być ostatni element listy, to wywołana zostanie `delete_at_back()` (wiersz 14). Niezależnie od tego, która z nich zostanie wywołana funkcja `delete_node()` po ich wykonaniu kończy swe działanie zwracając wartość wskaźnika `list_pointer`.

Wypisywanie zawartości listy

Wypisywanie w obu kierunkach

```
1  void print_list_in_both_directions(struct list_node
2                                                  *list_pointer)
3  {
4      struct list_node *backward_pointer = NULL;
5      while(list_pointer) {
6          backward_pointer = list_pointer;
7          printf("%d ",list_pointer->data);
8          list_pointer = list_pointer->next;
9      }
10     puts("");
11     while(backward_pointer) {
12         printf("%d ",backward_pointer->data);
13         backward_pointer = backward_pointer->previous;
14     }
15     puts("");
16 }
```

Wypisywanie zawartości listy

Funkcja, której kod źródłowy został zaprezentowany na poprzednim slajdzie wypisuje zawartość elementów listy w obu możliwych kierunkach, tj. od początku do końca i od końca do początku. Pierwszy przypadek jest wykonywany w pierwszej pętli `while` (wiersze 5 - 9). W tej pętli używany jest także wskaźnik `backward_pointer`, który poza jej pierwszą i ostatnią iteracją wskazuje na element poprzedzający element listy wskazywany wskaźnikiem `list_pointer`. Po zakończeniu pierwszej pętli `while` wskaźnik `backward_pointer` wskazuje na ostatni element listy. W drugiej pętli `while` ten wskaźnik jest używany do iterowania w odwrotnym kierunku po liście. Jego wartość w każdej jej iteracji jest zastępowana wartością pola `previous` elementu, na który on wskazuje.

Usuwanie listy

```
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  }
```

Usuwanie listy

Funkcja usuwająca listę z pamięci jest taka sama jak dla listy jednokierunkowej.

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_in_both_directions(list_pointer);
```

Funkcja `main()`

Część pierwsza

Podobnie jak w przypadku listy jednokierunkowej listę dwukierunkową tworzymy z najpierw jednym elementem o wartości 1, a następnie dodajemy do niej elementy o wartościach od 2 do 4 i od 6 do 9. Następnie zawartość listy jest dwukrotnie i dwukierunkowo wypisywana (wiersz nr 9).

Funkcja `main()`

Część druga

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

Funkcja `main()`

Część druga

Aby przetestować działanie funkcji `add_node()` w funkcji `main()` programu do dwukierunkowej listy liniowej dodawane są nowe elementy o wartościach odpowiednio `0` (dodanie na początek listy), `5` (dodanie wewnątrz listy), `7` (dodanie wewnątrz listy przed elementem o takiej samej wartości) i `10` (dodanie na koniec listy). Po wykonaniu każdej z tych operacji zawartość listy jest wypisywana dwukierunkowo na ekranie.

Funkcja main()

Część trzecia

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

Funkcja `main()`

Część trzecia

Podobnie jak w przypadku `add_node()`, aby przetestować działanie funkcji `delete_node()` z dwukierunkowej listy liniowej usuwane są elementy o wartościach `0` (początek listy), `1` (ponownie początek listy), `1` (nieistniejący element), `5` (wnętrze listy), `7` (pierwszy z dwóch elementów listy zawierających tę samą liczbę) oraz `10` (ostatni element listy). Po każdej takiej operacji zawartość listy jest wypisywana dwukierunkowo. Na zakończenie programu lista jest usuwana (wiersz nr 13) i kończy się działanie funkcji `main()`.

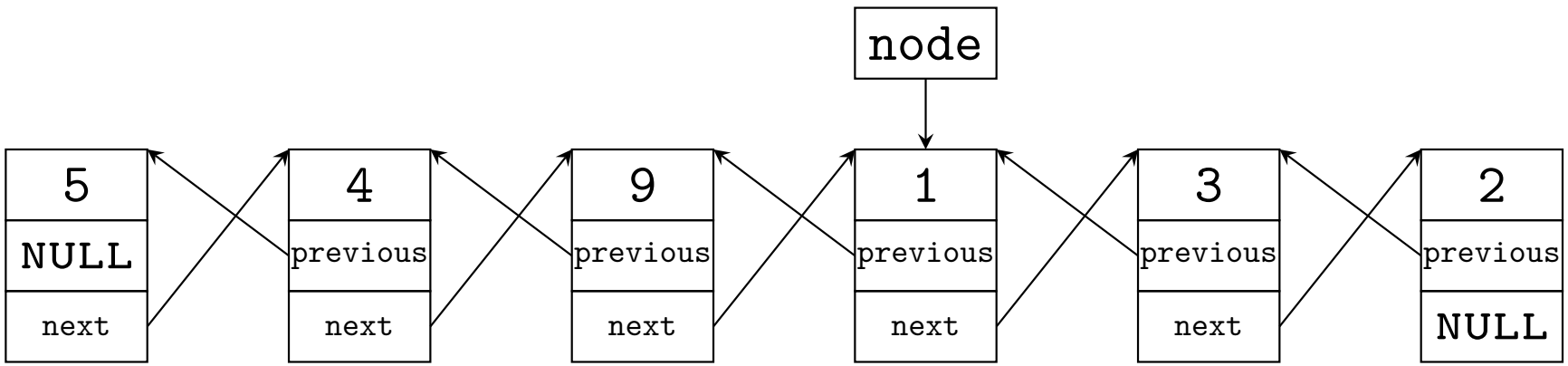
Podsumowanie

Przedstawiona implementacja dwukierunkowych list liniowych nie jest jedyną, jaką można zbudować. Dwukierunkowe listy liniowe mogą być zrealizowane również w oparciu o tablice jedno lub wielowymiarowe, podobnie jak listy jednokierunkowe. Istnieją również dwukierunkowe listy liniowe z wartownikami. Dynamiczne dwukierunkowe listy liniowe mogą posłużyć do budowy stosu lub kolejek. W pewnych zastosowaniach listy dwukierunkowe mają pewną przewagę nad listami jednokierunkowymi - podwójne powiązanie każdego z elementów. Ma to znaczenie np. w przypadku systemów plików, gdzie takie listy reprezentują plik. W takim przypadku są one tworzone nie w pamięci operacyjnej komputera lecz w pamięci masowej, np. na twardym dysku.

Podsumowanie

W zaprezentowanych funkcjach, takich jak `delete_in_middle()` lub `delete_at_back()` można zauważyć skomplikowane wyrażenia budowane przy użyciu wskaźników. Następne slajdy pokazują jeszcze bardziej skomplikowane przypadki takich wyrażeń. Odnoszą się one do przedstawionej w górnej części slajdów listy. Wskaźnik `node`, który występuje na początku każdego takiego wyrażenia również jest zamieszczony na podanej ilustracji. Proszę spróbować ustalić wartości tych wyrażeń.

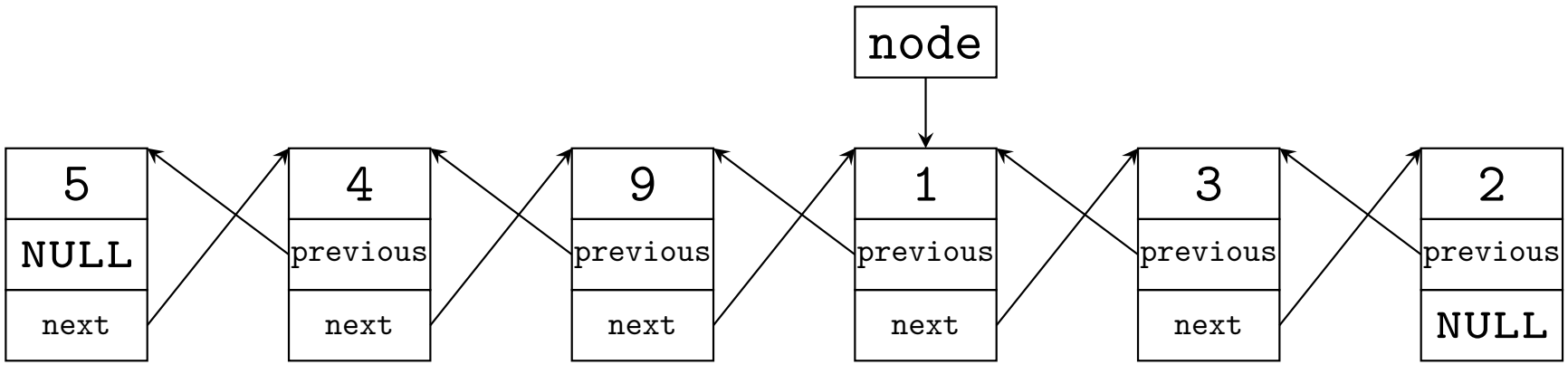
Podsumowanie



Wyrażenie nr 1

```
node->next->next->data
```

Podsumowanie



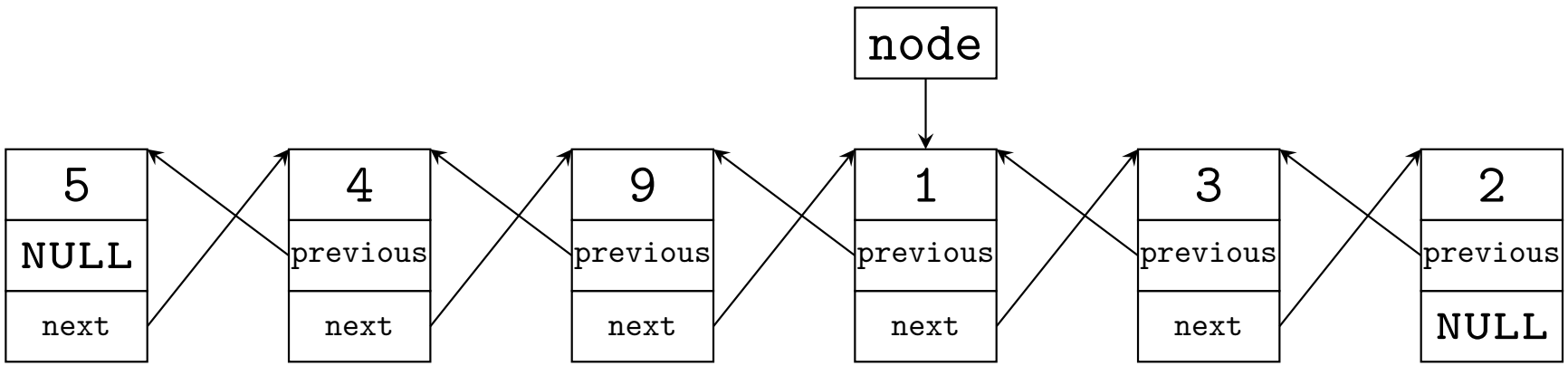
Wyrażenie nr 1

```
node->next->next->data
```

Odpowiedź nr 1

2

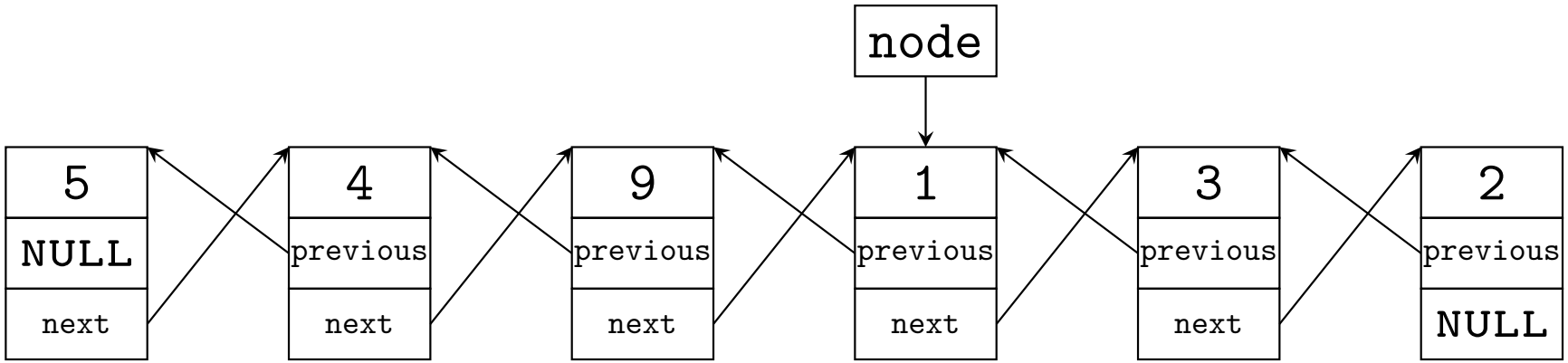
Podsumowanie



Wyrażenie nr 2

```
node->previous->previous->previous->data
```

Podsumowanie



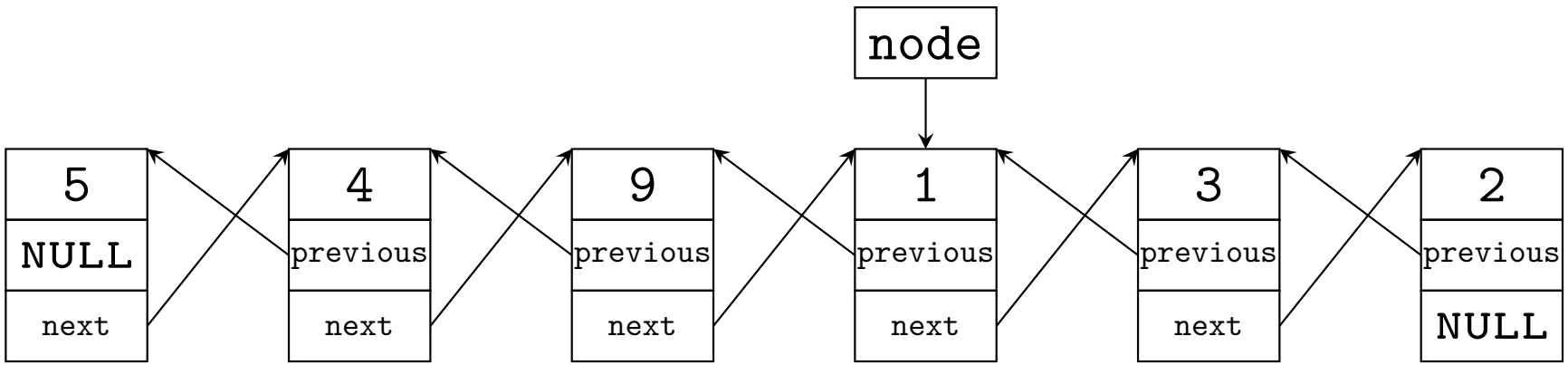
Wyrażenie nr 2

```
node->previous->previous->previous->data
```

Odpowiedź nr 2

5

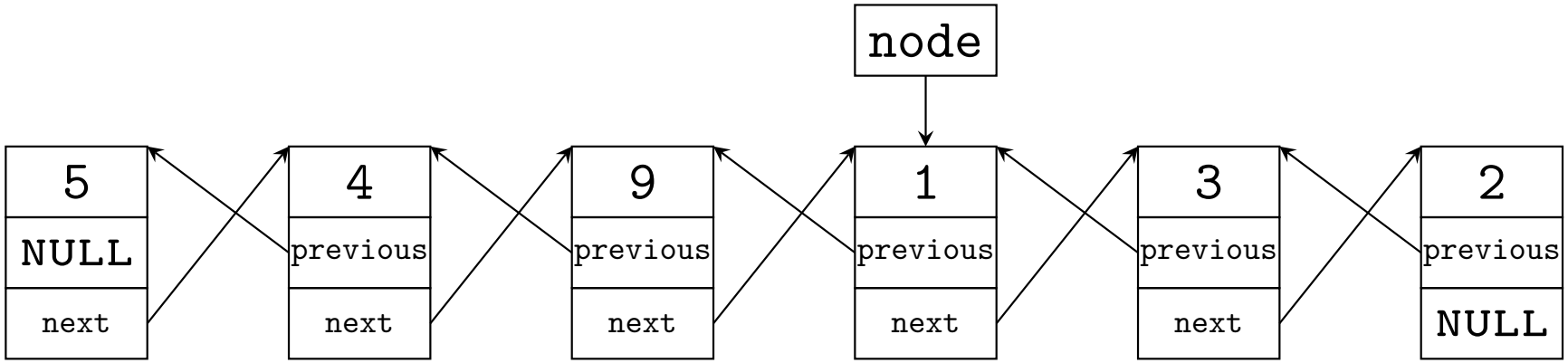
Podsumowanie



Wyrażenie nr 3

```
node->next->next->previous->previous->previous->previous->data
```

Podsumowanie



Wyrażenie nr 3

```
node->next->next->previous->previous->previous->previous->data
```

Odpowiedź nr 3

4

Podsumowanie

Reguła odczytu tych wyrażeń jest dosyć prosta - należy podążać za wskaźnikami. Warto jednak zwrócić uwagę na ostanie wyrażenie, gdzie wymieszane jest użycie wskaźników `next` i `previous`. Te wskaźniki wzajemnie „znoszą się". Zatem to wyrażenie można zapisać w skróconej formie jako: `node->previous->previous->data`.
Wniosek jaki nasuwa się po zapoznaniu się z tak rozbudowanymi wyrażeniami wskaźnikowymi jest następujący: Należy wiedzieć jak czytać takie wyrażenia i jak one działają, ale powinno unikać się stosowania ich w programach ☺.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!