Розділ 3.
Абстрактні типи даних

3.1. Визначення абстрактного типу даних

Література для самостійного читання:

с. 23-27 [1], с. 310-311 [4], с.47-84 [2], с.75-245 [3]

Тип даних

Структура даних

Абстрактний тип даних

Тип даних - у мовах програмування тип даних змінної позначає множину значень, які може приймати ця змінна.

Структура даних

Абстрактний тип даних

Тип даних - у мовах програмування тип даних змінної позначає множину значень, які може приймати ця змінна.

Структура даних

Абстрактний тип даних - це математична модель плюс різні оператори, визначені в рамках цієї моделі.

Тип даних - у мовах програмування тип даних змінної позначає множину значень, які може приймати ця змінна.

Структура даних - представлення АТД в термінах типів даних і операторів, підтримуваних даною мовою програмування.

Абстрактний тип даних - це математична модель плюс різні оператори, визначені в рамках цієї моделі.

Базовим будівельним блоком структури даних є комірка, яка призначена для зберігання значення певного базового або складеного типу даних.

Структури даних створюються шляхом задання імен сукупностям (агрегатам) комірок і (необов'язково) інтерпретації значення деяких комірок як представників (тобто покажчиків) інших комірок.

Способи агрегації комірок для створення структур даних:

- одновимірний масив
- запис
- файл
- покажчик + запис
- курсор + одновимірний масив

3.2. АТД "Список"

Література для самостійного читання:

с. 45-57 [1], с. 310-311 [4]

Приклад. Здійснюється реєстрація автомобілів, які прибувають на автостоянку та залишають її. Потрібно зберігати і обробляти множину номерів автомобілів. Для відображення цієї множини в пам'яті комп'ютера необхідно обрати певну структуру даних.

Лінійні зв’язні списки – це ефективна структура даних для моделювання ситуацій, в яких впорядкований масив даних треба змінювати.

- *Зв'язний лінійний список* — це сукупність однотипних компонентів, які послідовно зв'язані між собою за допомогою покажчиків.

Кожен компонент списку, крім останнього, містить покажчик на наступний (або на наступний і попередній) компонент.

Доступ до першого компонента здійснюється за допомогою покажчика на нього, а доступ до кожного наступного компонента — з використанням покажчика, який зберігається у попередньому компоненті.

Перший компонент списку називається його вершиною, або головою.

Різновиди однозв'язних списків:

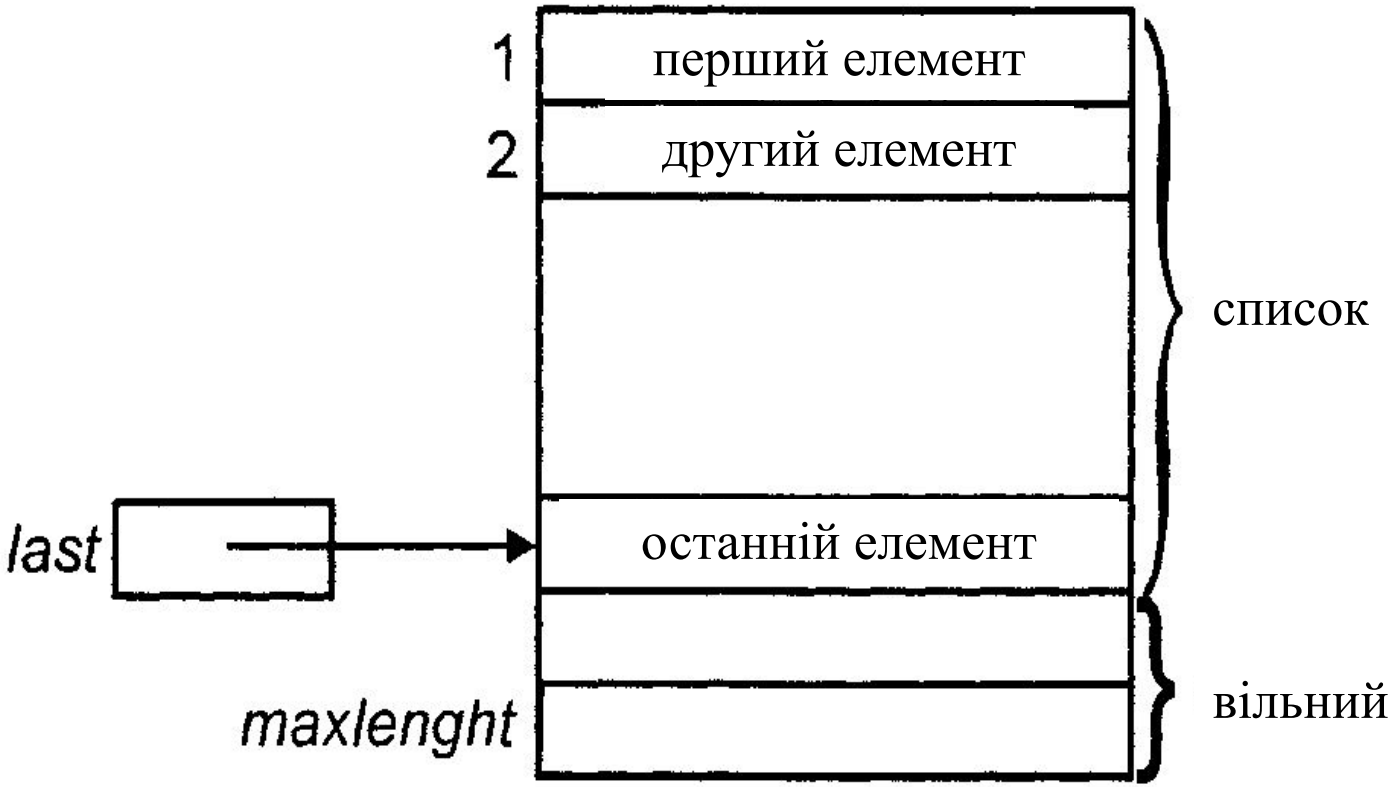
- ***Стек*** — це однозв'язний лінійний список, в якому компоненти додаються та видаляються лише з його вершини, тобто з початку списку.
- ***Черга*** — це однозв'язний лінійний список, в якому компоненти додаються в кінець списку, а видаляються з вершини, тобто з початку списку.
- ***Однозв'язний лінійний список*** — це список, в якому попередній компонент посилається на наступний.
- ***Однозв'язний циклічний список*** — це однозв'язний лінійний список, в якому останній компонент посилається на перший.

Різновиди двозв'язних списків:

- ***Двозв'язний лінійний список*** — це список, в якому попередній компонент посилається на наступний, а наступний — на попередній.
- ***Двозв'язний циклічний список*** — це двозв'язний лінійний список, в якому останній компонент посилається на перший, а перший компонент — на останній.

Реалізація списків за допомогою масивів

При реалізації списків за допомогою масивів елементи списку розташовуються в суміжних комірках масиву. Це уявлення дозволяє легко проглядати вміст списку і вставляти нові елементи в його кінець. Але вставка нового елементу в середину списку вимагає переміщення всіх подальших елементів на одну позицію до кінця масиву, щоб звільнити місце для нового елементу. Видалення елементу також вимагає переміщення елементів, щоб закрити комірку, що звільнилася.

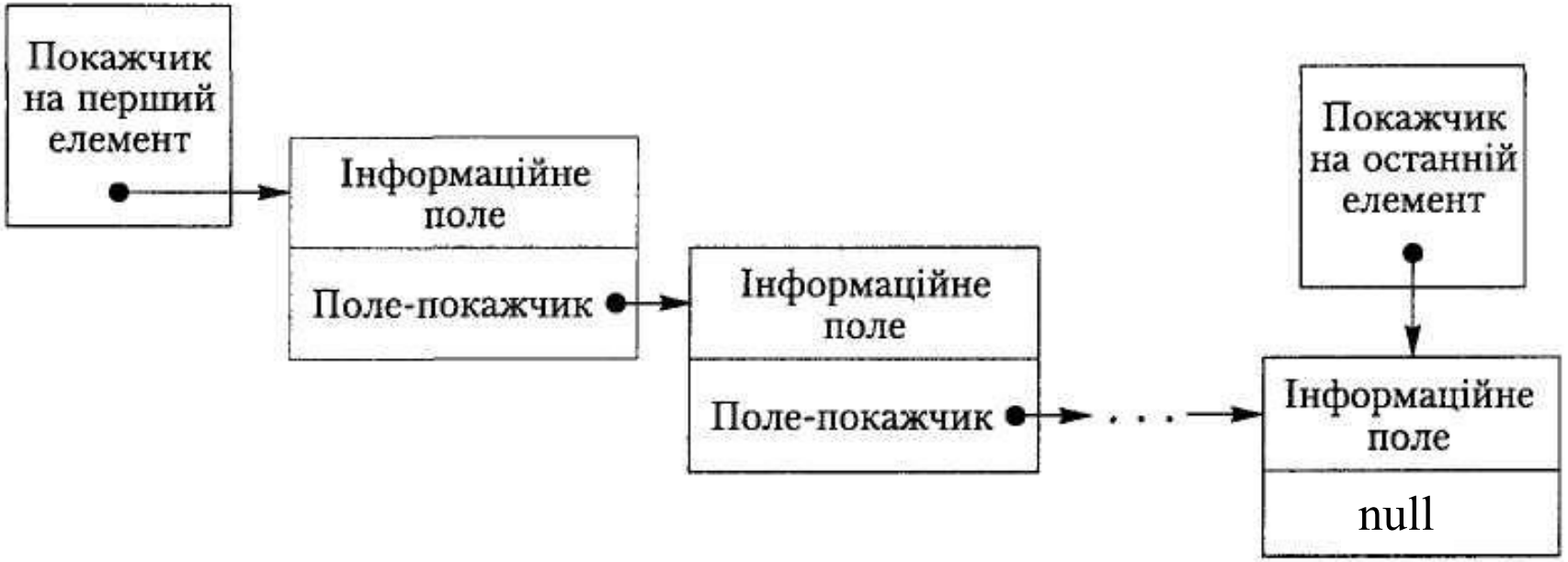


З прикладом реалізації можна ознайомитись в (с.48-50 [1]).

Реалізація списків за допомогою покажчиків

Кожний компонент зв'язного лінійного списку складається з кількох інформаційних полів та покажчика на наступний компонент. Отже, *компонент зв'язного лінійного списку є записом*. Інформаційні поля компонента списку можуть бути змінними будь-яких типів, а покажчик повинен бути покажчиком на запис того типу, якому належать компоненти списку. Покажчик в останньому компоненті лінійного списку має значення null (nil) — так позначається кінець списку.

Зображення однозв'язного лінійного списку



Приклад. Оголошення типу компонента однозв'язного лінійного списку в Pascal. Для роботи з таким списком потрібні покажчики на перший і поточний компоненти.

```
type
  ptr=^Item;      {тип покажчика на
                компонент списку}
  Item=record     {тип компонента}
   data  : string;   {інформаційне поле}
   next  : ptr; {покажчик на наступний
   end;           компонент}
var head,        {покажчики на перший та}
    current : ptr;  {поточний компоненти
                списку}
```

Приклад. Оголошення типу компонента однозв'язного лінійного списку в С.

```
// Декларація типу компонента
struct list {
   char data[25];     // інформаційне поле
   list *next_item;  // покажчик на наступний
                     компонент
};

// Декларація (глобальні змінні) покажчиків
struct list *head = NULL; // на перший
struct list *last = NULL; // на останній
```

Приклади, що ілюструють реалізації АТД “Список”:

– реалізація за допомогою покажчиків (с.50-53 [1]).
– реалізація за допомогою масивів (с.48-50 [1]).
– реалізація на основі курсорів (с.54-56 [1]).

Порівняння реалізацій АТД “Список”

Зрозуміло, нас не може не цікавити питання про те, в яких ситуаціях краще використовувати реалізацію списків за допомогою покажчиків, а коли - за допомогою масивів.

Відповідь на це питання залежить від того, які оператори повинні виконуватися над списками і як часто вони використовуватимуться. Іноді аргументом на користь однієї або іншої реалізації може служити максимальний розмір списків, що обробляються.

1. Реалізація списків за допомогою масивів вимагає вказівки максимального розміру списку до початку виконання програм.

   Якщо не можемо заздалегідь обмежити зверху довжину оброблюваних списків, то, очевидно, більш раціональним вибором буде реалізація списків за допомогою покажчиків.

2. Виконання деяких операторів в одній реалізації вимагає більших обчислювальних витрат, ніж в іншій.

   Наприклад, процедури INSERT і DELETE виконуються за постійне число кроків у разі зв'язних списків будь-якого розміру, але вимагають часу, пропорційного числу елементів, наступних за елементом, що вставляється (або що видаляється), при використанні масивів. І навпаки, час виконання функцій PREVIOUS і END постійний при реалізації списків за допомогою масивів, але цей же час пропорційний довжині списку у разі реалізації, побудованої за допомогою покажчиків.

3. Якщо необхідно вставляти або видаляти елементи, положення яких вказане з допомогою якоїсь змінної-курсору, і значення цієї змінної буде використане пізніше, то недоцільно використовувати реалізацію з допомогою покажчиків, оскільки ця змінна не "відстежує" вставку і видалення елементів.

   Використання покажчиків вимагає особливої уваги і ретельності в роботі.

4. Реалізація списків за допомогою масивів марнотратна відносно комп’ютерної пам'яті, оскільки резервується об'єм пам'яті, достатній для максимально можливого розміру списку незалежно від його реального розміру в конкретний момент часу.

   Реалізація за допомогою покажчиків використовує стільки пам'яті, скільки необхідно для зберігання поточного списку, але вимагає додаткову пам'ять для покажчика кожного запису.

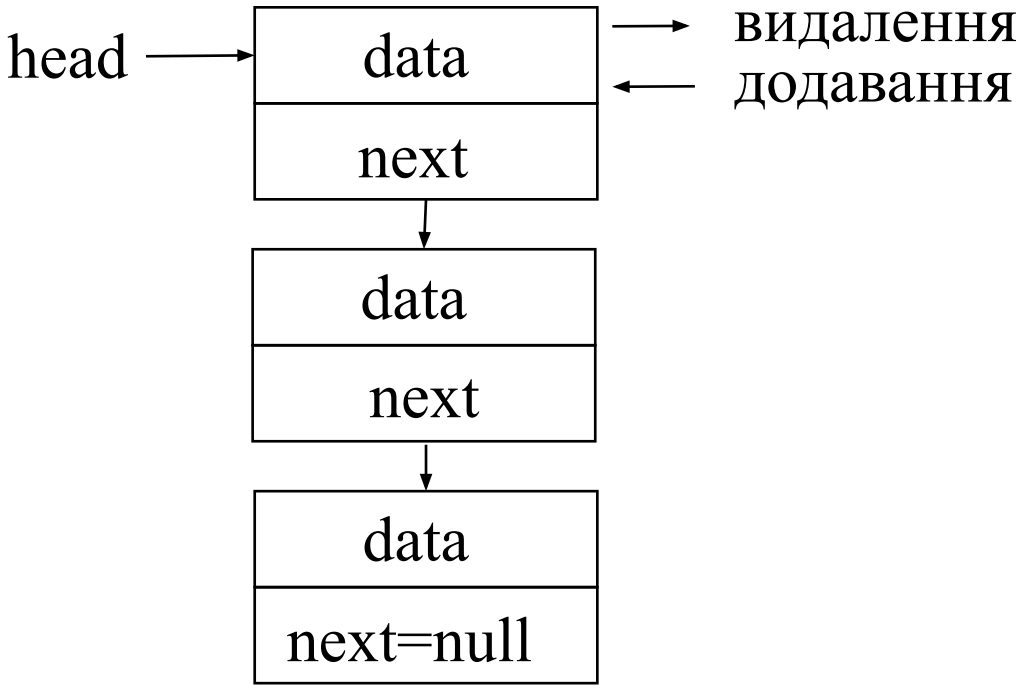
   В різних ситуаціях по критерію використаної пам'яті можуть бути вигідні різні реалізації.

3.3. Стек

Література для самостійного читання:

с. 58-60 [1], с. 312-316 [4]

Стек — це один із різновидів однозв'язного лінійного списку, доступ до елементів якого можливий лише через його початок, що називається *вершиною стеку.*



Для роботи зі стеком використовують зазвичай п’ять дій:

- перевірка, чи порожній стек
- додавання елемента у вершину стеку
- зчитування елементу у вершині стеку
- видалення елемента з вершини стеку
- очищення стеку

Реалізація стеків за допомогою покажчиків

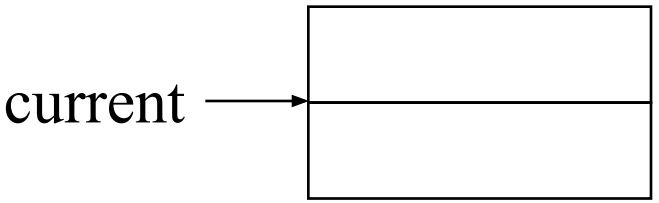
Стек працює за принципом «останнім прийшов — першим вийшов», що позначається абревіатурою LIFO (від англ. Last In First Out), і має такі властивості:

- елементи додаються у вершину (голову) стеку;
- елементи видаляються з вершини (голови) стеку;
- покажчик в останньому елементі дорівнює `null`;
- неможливо вилучити елемент із середини стеку, не вилучивши всі елементи, що йдуть попереду.

Для роботи зі стеком достатньо мати покажчик `head` на його вершину та допоміжний покажчик `current` на елемент стеку.

Алгоритм вставки елемента до порожнього стеку

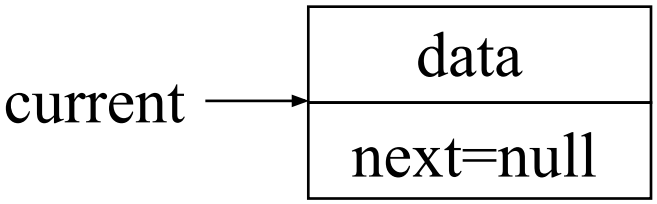
1. Виділити пам'ять для нового елемента стеку



Алгоритм вставки елемента до порожнього стеку

1. Виділити пам'ять для нового елемента стеку

2. Ввести дані до нового елемента

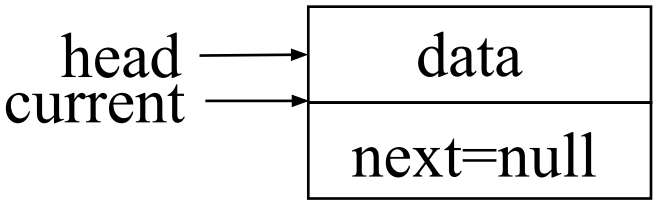


Алгоритм вставки елемента до порожнього стеку

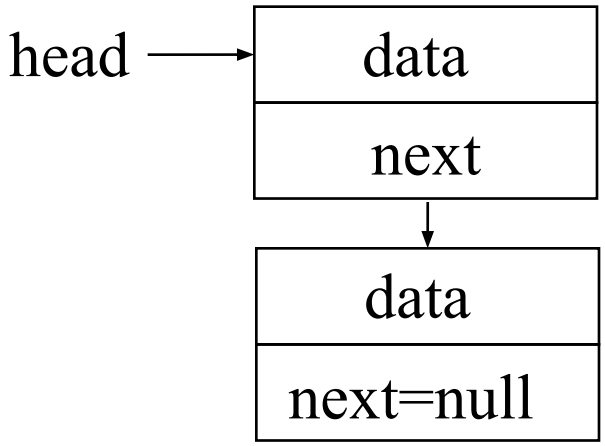
1. Виділити пам'ять для нового елемента стеку

2. Ввести дані до нового елемента

3. Встановити вершину стеку на новостворений елемент

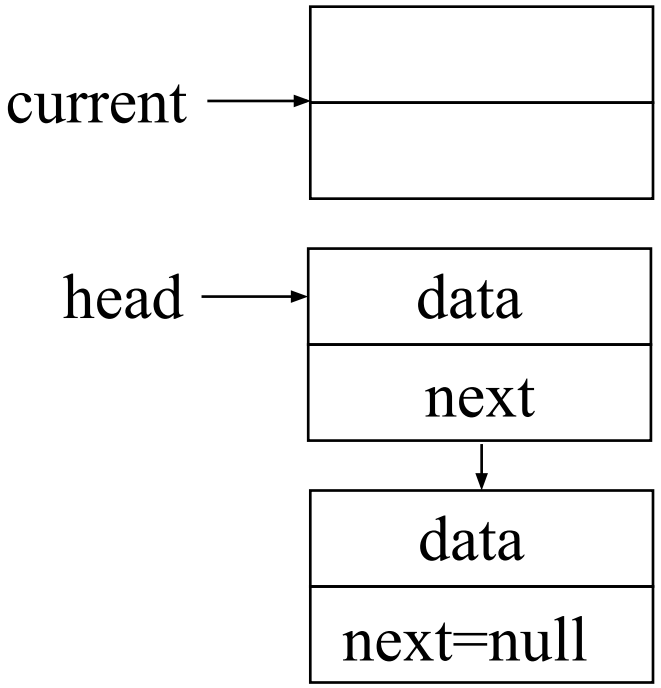


Алгоритм вставки елемента до стеку



Алгоритм вставки елемента до стеку

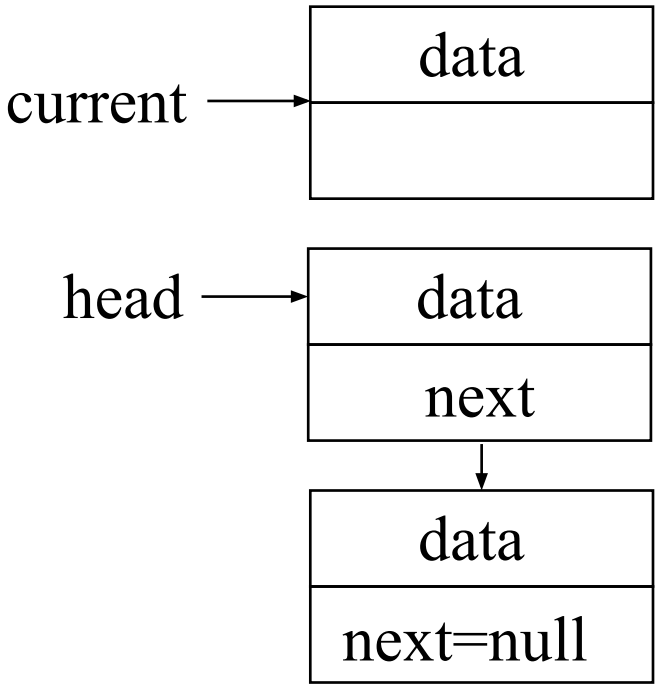
1. Виділити пам'ять для нового елемента стеку



Алгоритм вставки елемента до стеку

1. Виділити пам'ять для нового елемента стеку

2. Ввести дані до нового елемента

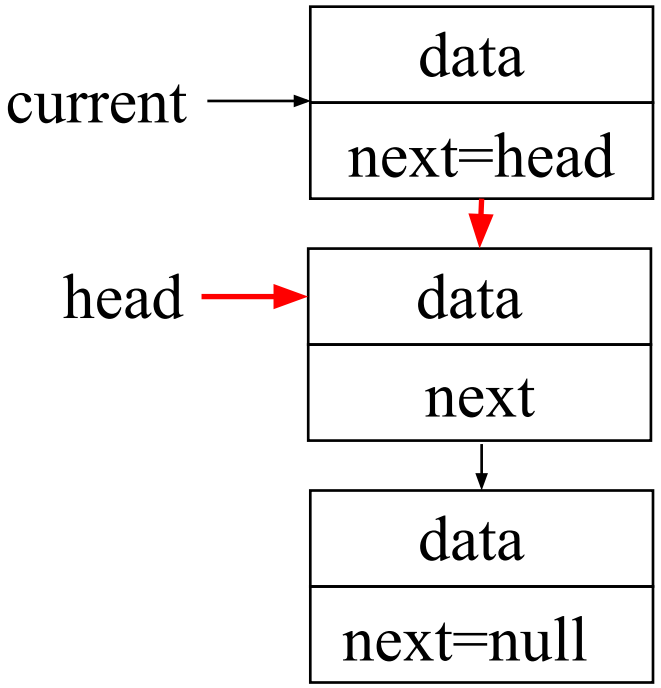


Алгоритм вставки елемента до стеку

1. Виділити пам'ять для нового елемента стеку

2. Ввести дані до нового елемента

3. Зв'язати новий елемент із вершиною



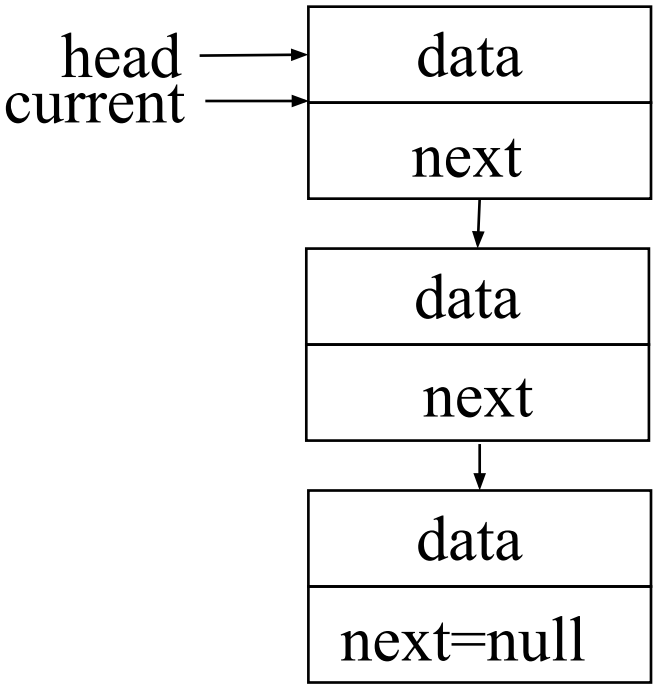
Алгоритм вставки елемента до стеку

1. Виділити пам'ять для нового елемента стеку

2. Ввести дані до нового елемента

3. Зв'язати новий елемент із вершиною

4. Встановити вершину стеку на новостворений елемент



Опис структури даних

Оголошення типу компонента однозв'язного лінійного списку.

```
// Декларація типу компонента
struct list {
   char    data[25];    // інформаційне поле
   list   *next_item;   // покажчик на наступний компонент
};

// Декларація покажчика
struct list *head;     // покажчики на перший
struct list *current;  // на поточний компонент
```

Встановлення початковых значень

0) початкове значення змінних.

Обидві змінні містять невизначене значення.

current = head = NULL;

```
129 int main()
130 {
131    // Декларація покажчика
132    struct list *current;  // поточний компоненти черги
133    struct list *head;     // покажчики на перший
134    // 0
135    current = head = NULL;
136    // 1.  current <неопределенное значение>
137    // 1.1. Виділити пам'ять для нового елемента стеку:
138    current = new struct list;
139    // 1.2. Ввести дані до нового елемента
140    cout << "Input value for data field\n";
141    cin >> current->data;
142    // 1.3. Зв'язати допоміжний елемент із вершиною
143    current->next_item = head;
144    // 1.4. Встановити вершину стеку на новостворений елемент
145    head = current;
```

100 %

Локальные

Имя	Значение
head	<неопределенное значение>
current	<неопределенное значение>

Створення першого елементу списку
1.1) здійснено виділення пам’яті
current = new struct list;

```
129 int main()
130 {
131     // Декларація покажчика
132     struct list *current;  // поточний компоненти черги
133     struct list *head;     // покажчики на перший
134     // 0
135     current = head = NULL;
136     // 1. Створення першого елементу списку
137     // 1.1. Виділити пам'ять для нового елемента стеку:
138     current = new struct list;
139     // 1.1  current 0x003B6298 { data="HHHHHHHHHHHHHHHHHHHHHHHHH" next_item=0xCDCDCDCD }
140     memset(cu   data "HHHHHHHHHHHHHHHHHHHHHHHHH"
141     // 1.2. B   next_item 0xCDCDCDCD
142     cout << "Input value for data field\n";
143     cin >> current->data;
144     // 1.3. Зв'язати допоміжний елемент із вершиною
145     current->next_item = head;
```

100 %

Локальные

Имя	Значение
head	<неопределенное значение>
current	0x003B6298 { data="HHHHHHHHHHHHHHHHHHHHHHHHH" next_item=0xCDCDCDCD }

Пам'ять містить «сміття». Виконаємо занулення виділеної пам’яті.

1.1.1. Занулення

memset(current, 0,sizeof(struct list));

```
129 int main()
130 {
131    // Декларація покажчика
132    struct list *current;  // поточний компоненти черги
133    struct list *head;     // покажчики на перший
134    // 0
135    current = head = NULL;
136    // 1. Створення першого елементу списку
137    // 1.1. Виділити пам'ять для нового елемента стеку:
138    current = new struct list;
139    // 1.1.1. Занулення пам'яті (не обовязковий крок)
140    memset(current, 0,sizeof(struct list));
141    // 1.2. Вв  current 0x003B6298 { data="" next_item=0x00000000 }
142    cout << "Inpu   data  ""
143    cin >> curren  next_item  0x00000000
144    // 1.3. Зв'язати допоміжний елемент із вершиною
145    current->next_item = head;
```

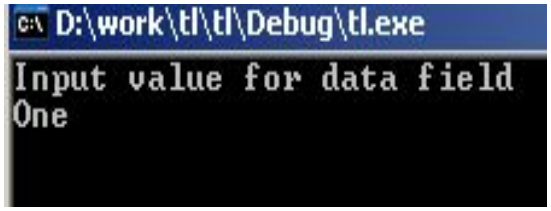
100 %

Локальные

Имя	Значение
head	<неопределенное значение>
current	0x003B6298 { data="" next_item=0x00000000 }
data	""
next_item	0x00000000

1.2. Здійснюємо ввод даних із клавіатури:

```
cout << "Input value for data field\n";
cin >> current->data;
```




```
135    current = head = NULL;
136    // 1. Створення першого елементу списку
137    // 1.1. Виділити пам'ять для нового елемента стеку:
138    current = new struct list;
139    // 1.1.1. Занулення пам'яті (не обовязковий крок)
140    memset(current, 0,sizeof(struct list));
141    // 1.2. Ввести дані до нового елемента
142    cout << "Input value for data field\n";
143    cin >> current->data;
144    // 1.3. Зв'
145    current->next
146    // 1.4. Встан                створений елемент
147    head = current;
148
149    // 2. Створення першого елементу списку
150    // 2.1. Виділити пам'ять для нового елемента стеку:
151    current = new struct list;
```

current 0x003B6298 { data="One" next_item=0x00000000 }
data "One"
next_item 0x00000000

.00 %

окальные

Имя	Значение
head	<неопределенное значение>
current	0x003B6298 { data="One" next_item=0x00000000 }
data	"One"
next_item	0x00000000

1.3. Зв'язати допоміжний елемент із вершиною:

current->next_item = head;

Оскільки мали порожнє значення в елементі HEAD, то візуально зміни не спостерігаються.

```
135    current = head = NULL;
136    // 1. Створення першого елементу списку
137    // 1.1. Виділити пам'ять для нового елемента стеку:
138    current = new struct list;
139    // 1.1.1. Занулення пам'яті (не обовязковий крок)
140    memset(current, 0,sizeof(struct list));
141    // 1.2. Ввести дані до нового елемента
142    cout << "Input value for data field\n";
143    cin >> current->data;
144    // 1.3. Зв'язати допоміжний елемент із вершиною
145    current->next_item = head;
146    //  current 0x003B6298 { data="One" next_item=0x00000000 }
147    head =
148
149    // 2. Створення першого елементу списку
150    // 2.1. Виділити пам'ять для нового елемента стеку:
151    current = new struct list;
```

data "One"

next_item 0x00000000

100 %

Локальные

Имя	Значение
head	<неопределенное значение>
current	0x003B6298 { data="One" next_item=0x00000000 }
data	"One"
next_item	0x00000000

1.4. Встановити вершину стеку на новостворений елемент

head = current;

```
135    current = head = NULL;
136    // 1. Створення першого елементу списку
137    // 1.1. Виділити пам'ять для нового елемента стеку:
138    current = new struct list;
139    // 1.1.1. Занулення пам'яті (не обовязковий крок)
140    memset(current, 0,sizeof(struct list));
141    // 1.2. Ввести дані до нового елемента
142    cout << "Input value for data field\n";
143    cin >> current->data;
144    // 1.3. Зв'язати допоміжний елемент із вершиною
145    current->next_item = head;
146    // 1.4. Встановити вершину стеку на новостворений елемент
147    head = current;
148
149    // 2.                                    списку
150    // 2.                                 ого елемента стеку:
151    current = new struct list;
```

head 0x003B6298 { data="One" next_item=0x00000000 }

data "One"

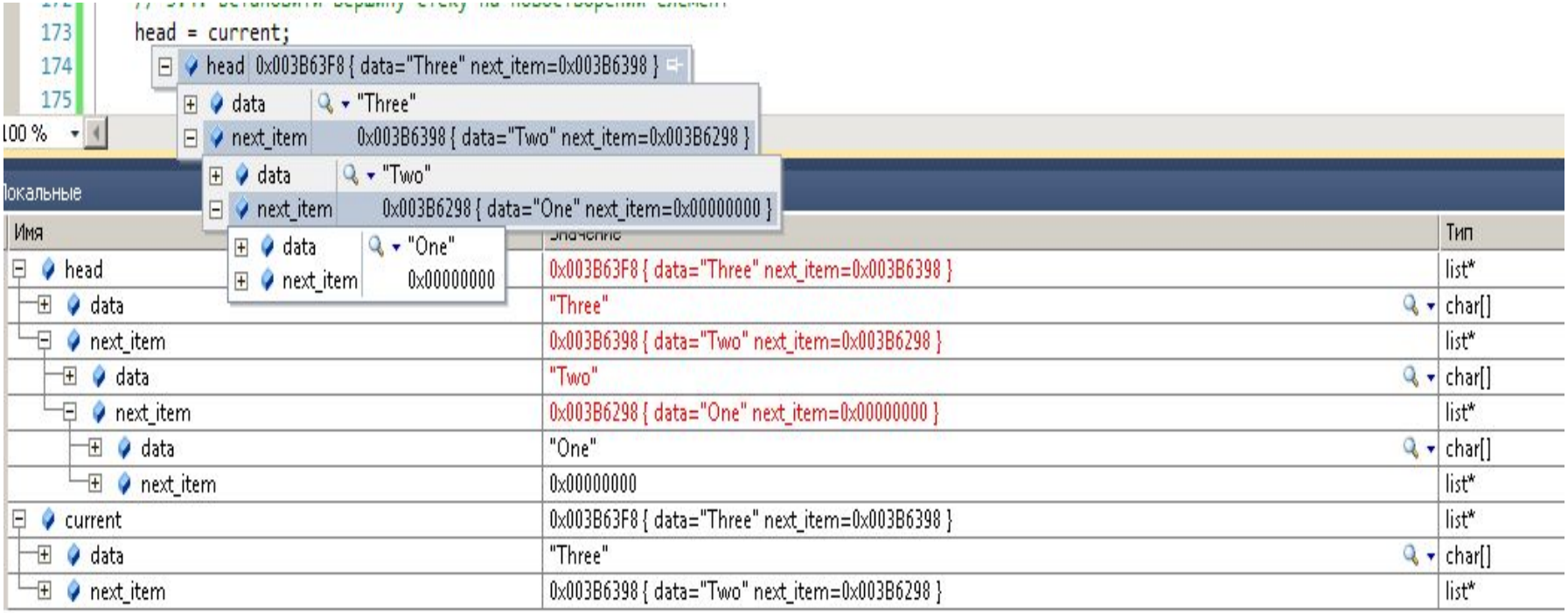
next_item 0x00000000

100 %

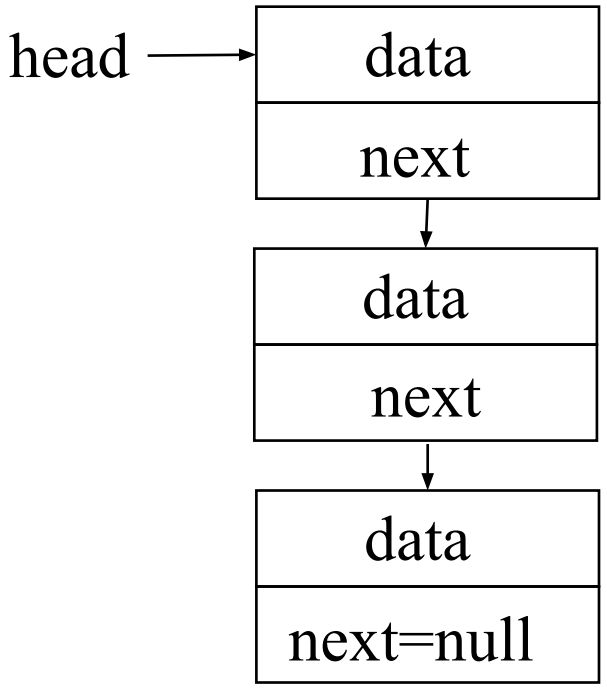
Локальные

Имя	Значение
head	0x003B6298 { data="One" next_item=0x00000000 }
current	0x003B6298 { data="One" next_item=0x00000000 }
data	"One"
next_item	0x00000000

В стек записано три елементи

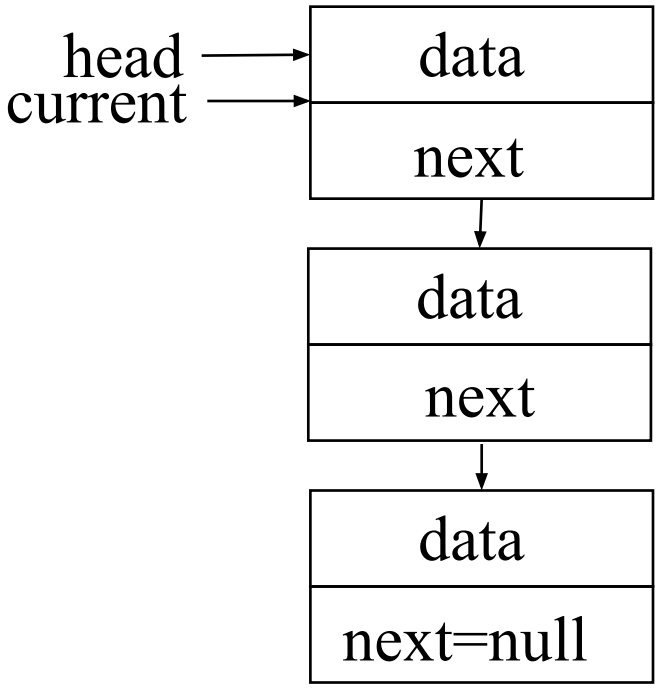


Алгоритм видалення елемента зі стеку



Алгоритм видалення елемента зі стеку

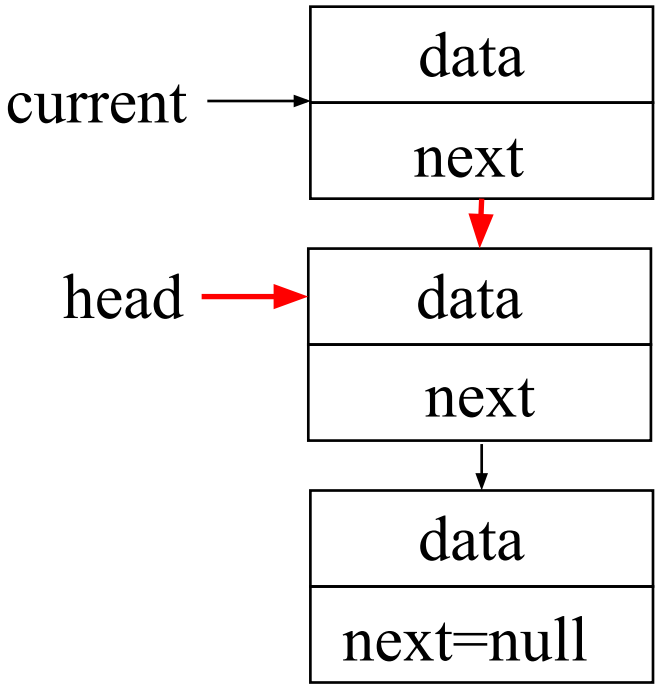
1. Створити копію покажчика на вершину стеку



Алгоритм видалення елемента зі стеку

1. Створити копію покажчика на вершину стеку

2. Перемістити покажчик на вершину стеку на наступний елемент



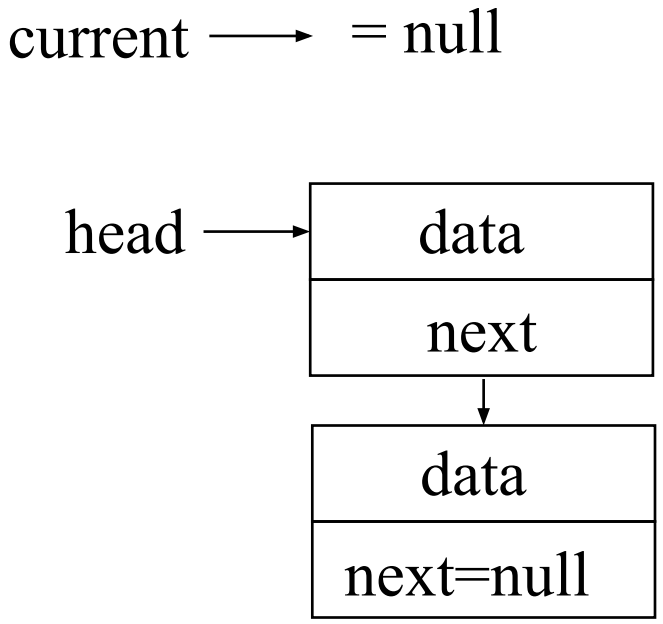
Алгоритм видалення елемента зі стеку

1. Створити копію покажчика на вершину стеку

2. Перемістити покажчик на вершину стеку на наступний елемент

3. Звільнити пам'ять із-під колишньої вершини стеку

4. Для очищення всього стеку слід повторювати кроки 1-3 доти, доки покажчик `head` не дорівнюватиме `null`.

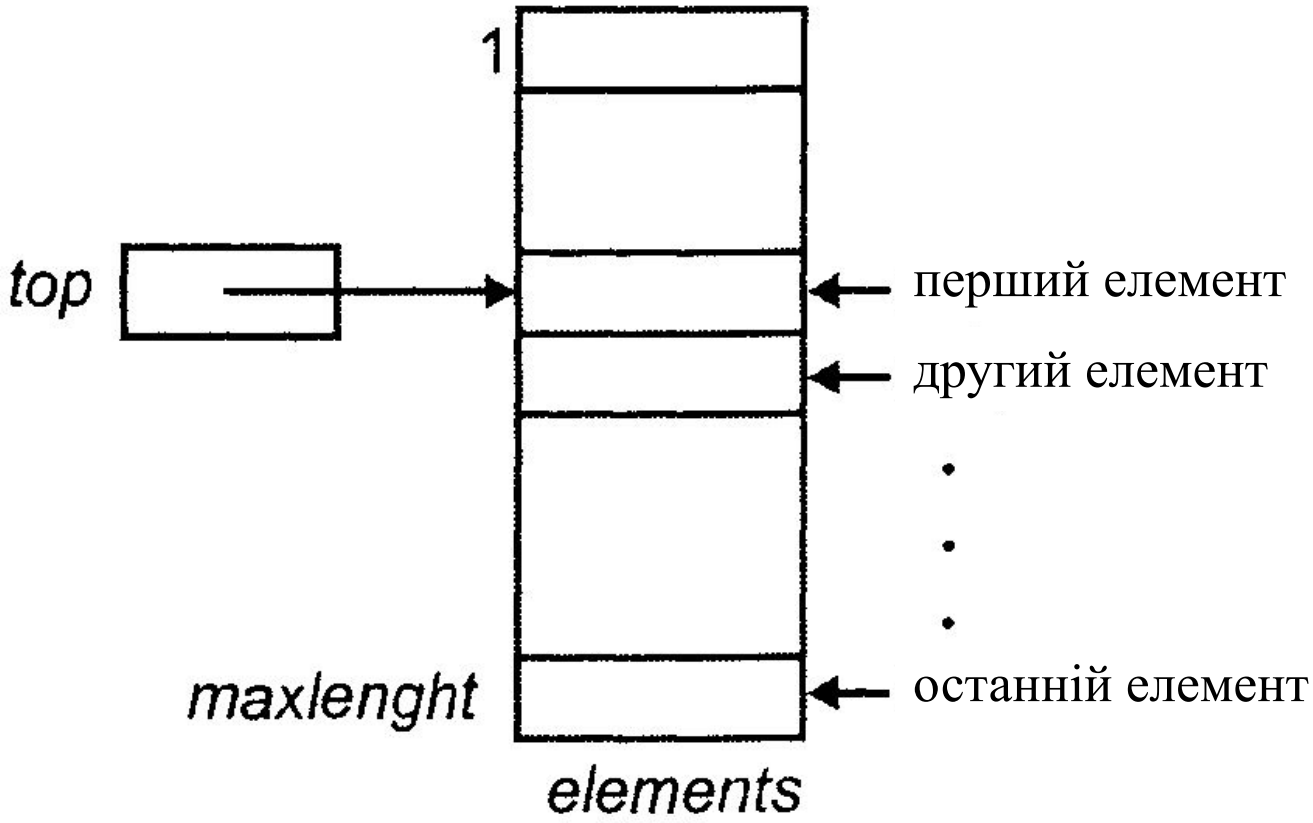


Реалізація стеків за допомогою масивів

Кожну реалізацію списків можна розглядати як реалізацію стеків, оскільки стеки з їх операторами є окремими випадками списків з операторами, що виконуються над списками.

Проте реалізація списків на основі масивів, описана раніше, не дуже підходить для представлення стеків, оскільки кожне виконання операторів додавання і видалення елемента в цьому випадку вимагає переміщення всіх елементів стека і тому час їх виконання пропорційний числу елементів в стеку.

Можна раціональніше пристосувати масиви для реалізації стеків, якщо взяти до уваги той факт, що вставка і видалення елементів стека відбувається тільки через вершину стека. Можна зафіксувати "дно" стека в самому низу масиву (у записі з найбільшим індексом) і дозволити стеку рости вгору масиву (до запису з найменшим індексом). Курсор з ім'ям top (вершина) вказуватиме положення поточної позиції першого елементу стека.



З прикладом реалізації можна ознайомитись в (с.60-61 [1]).

Приклади, що ілюструють реалізації АТД “Стек”:

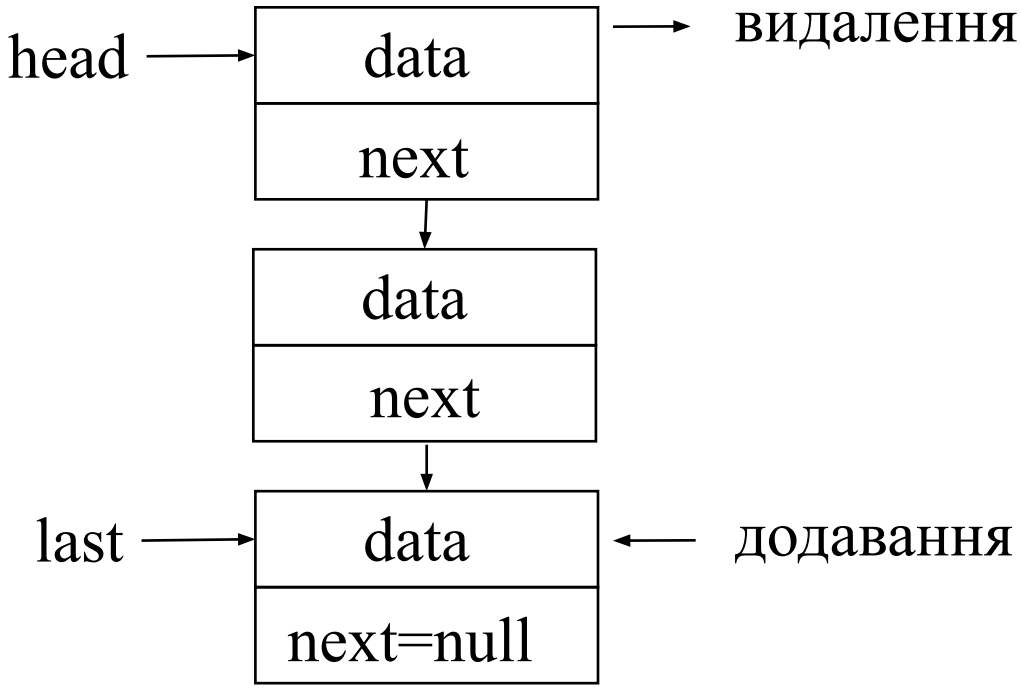
– реалізація за допомогою покажчиків (с.310-315 [4])
– ще одна реалізація за допомогою покажчиків (с.58-60 [1])
– реалізація за допомогою масивів (с.60-61 [1]).

3.4. Черга

Література для самостійного читання:

с. 57-65 [1], с. 316-325 [4]

Черга, як і стек, — це один із різновидів однозв'язного лінійного списку.



Для роботи з чергою використовують такі дії:

– перевірка, чи порожня черга

– додавання елемента з кінець черги

– зчитування елементу з початку черги

– видалення елемента з початку черги

– очищення черги

Реалізація черг за допомогою покажчиків

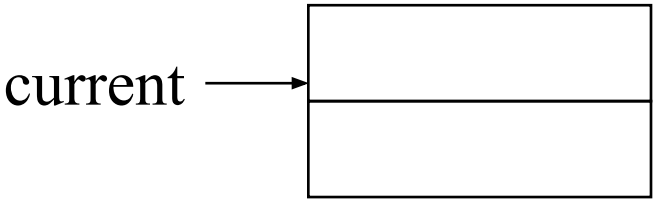
Черга працює за принципом «першим прийшов — першим вийшов», що позначається абревіатурою FIFO (від англ. First In First Out), і характеризується такими властивостями:

- елементи додаються в кінець черги;
- елементи зчитуються та видаляються з початку (вершини) черги;
- покажчик в останньому елементі дорівнює `null`;
- неможливо отримати елемент із середини черги, не вилучивши всі елементи, що йдуть попереду.

Для роботи з чергою потрібні: покажчик `head` на початок черги, покажчик `last` на кінець черги та допоміжний покажчик `current`.

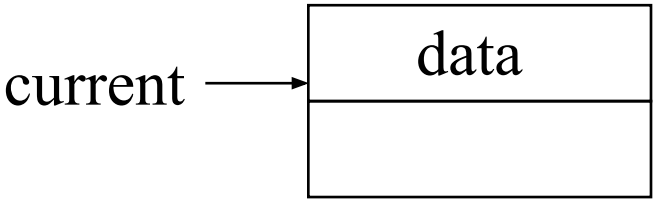
Алгоритм вставки елемента до порожньої черги

1. Виділити пам'ять для нового
   елемента черги



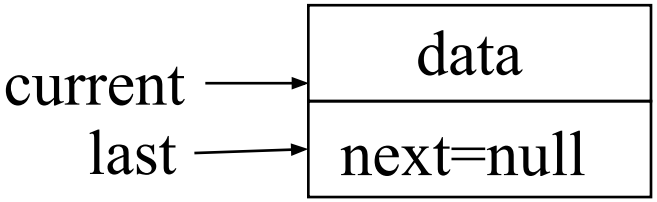
Алгоритм вставки елемента до порожньої черги

1. Виділити пам'ять для нового
   елемента черги
2. Ввести дані до нового елемента



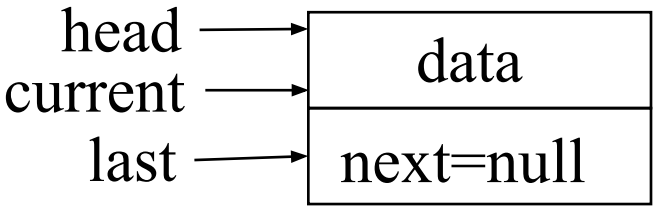
Алгоритм вставки елемента до порожньої черги

1. Виділити пам'ять для нового елемента черги
2. Ввести дані до нового елемента
3. Вважати новий елемент останнім у черзі



Алгоритм вставки елемента до порожньої черги

1. Виділити пам'ять для нового елемента черги
2. Ввести дані до нового елемента
3. Вважати новий елемент останнім у черзі
4. Якщо черга порожня, то вважати новий елемент вершиною черги



Опис структури даних

Оголошення типу компонента однозв'язного лінійного списку.

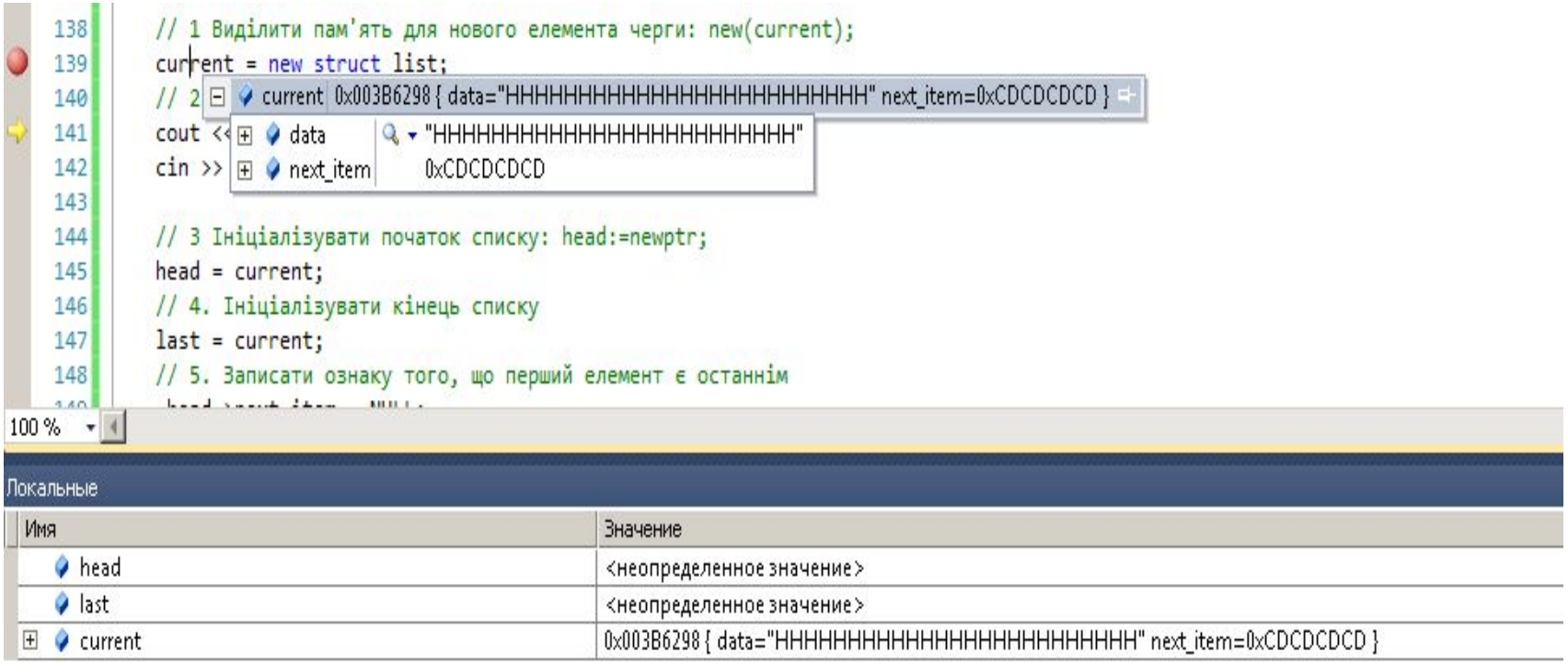
```
// Декларація типу компонента
struct list {
   char    data[25];    // інформаційне поле
   list   *next_item;   // покажчик на наступний компонент
};

// Декларація покажчика
struct list *head;      // покажчики на перший
struct list *current;  // поточний компонент черги
struct list *last;      // останній елемент черги
```

Створення елементу списку. Введення даних в елемент.

```
struct list *add_to_list(struct list *p_head)
{ struct list *current = NULL;  // поточний компоненти черги
  struct list *last    = NULL;     // останній елемент черги
  // Виділити пам'ять для нового елемента черги
  current = new struct list;
  // Ввести дані до нового елемента
  cout << "Input value for data field\n";
  cin >> current->data;
  // Вважати новий елемент останнім у черзі
  current->next_item = NULL;
```

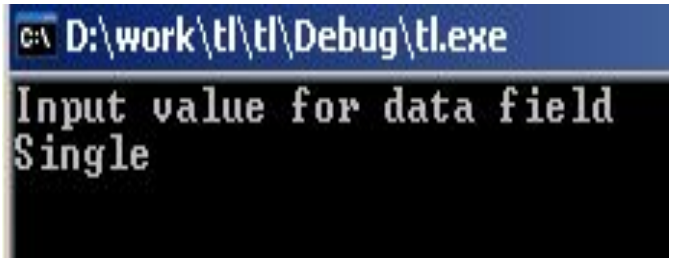
```
  //Якщо черга порожня, то ініціалізувати її вершину
  if (p_head == NULL)
 {p_head = current;}
  else
  { // Якщо черга не порожня, то зв'язати останній
    елемент черги із новоутвореним
    // 1. визначення останнього елемента черги
    last = p_head;
    while (last->next_item != NULL)
    {last = last->next_item; }
    // 2.   зв'язати останній елемент черги із новоутвореним
    last->next_item = current;
  }
  return p_head;
}
```

1) початкове значення змінних



2) ввод даних із клавіатури

```
cout << "Input value for data field\n";
cin >> current->data;
```




```
141    cout << "Input value for data field\n";
142    cin >> current->data;
143                current 0x003B6298 { data="Single" next_item=0xCDCDCDCD }
144    // 3 Ініціа  data  "Single"          :=newptr;
145    head = curr  next_item  0xCDCDCDCD
146    // 4. Ініціалізувати кінець списку
147    last = current;
148    // 5. Записати ознаку того, що перший елемент є останнім
149     head->next_item = NULL;
150
151
152
153
154    // Декларація покажчика
```

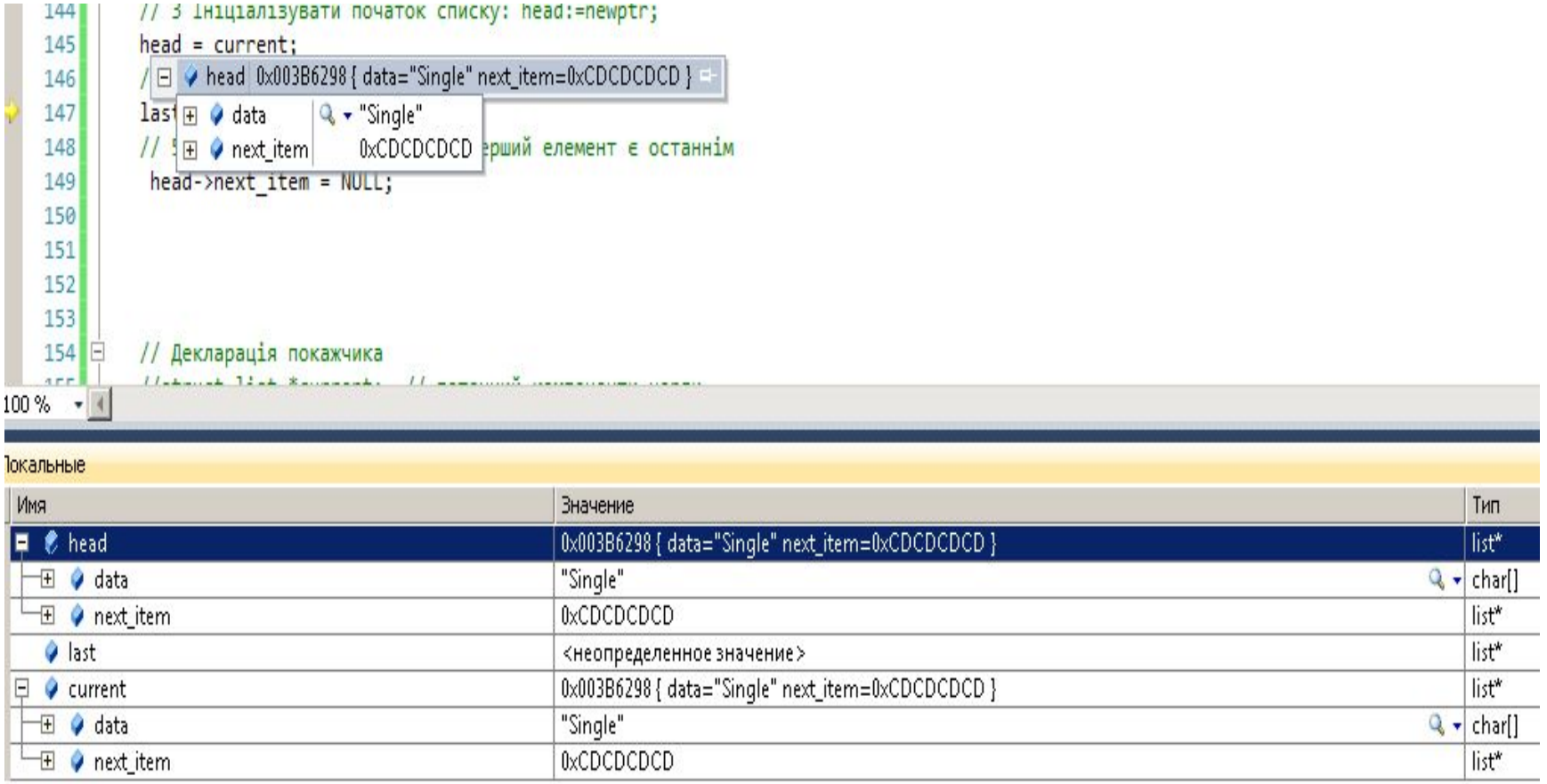
100 %

Локальные

Имя	Значение	Тип
head	<неопределенное значение>	list*
last	<неопределенное значение>	list*
current	0x003B6298 { data="Single" next_item=0xCDCDCDCD }	list*
data	"Single"	char[]
next_item	0xCDCDCDCD	list*

3) Ініціалізувати початок списку

head = current;

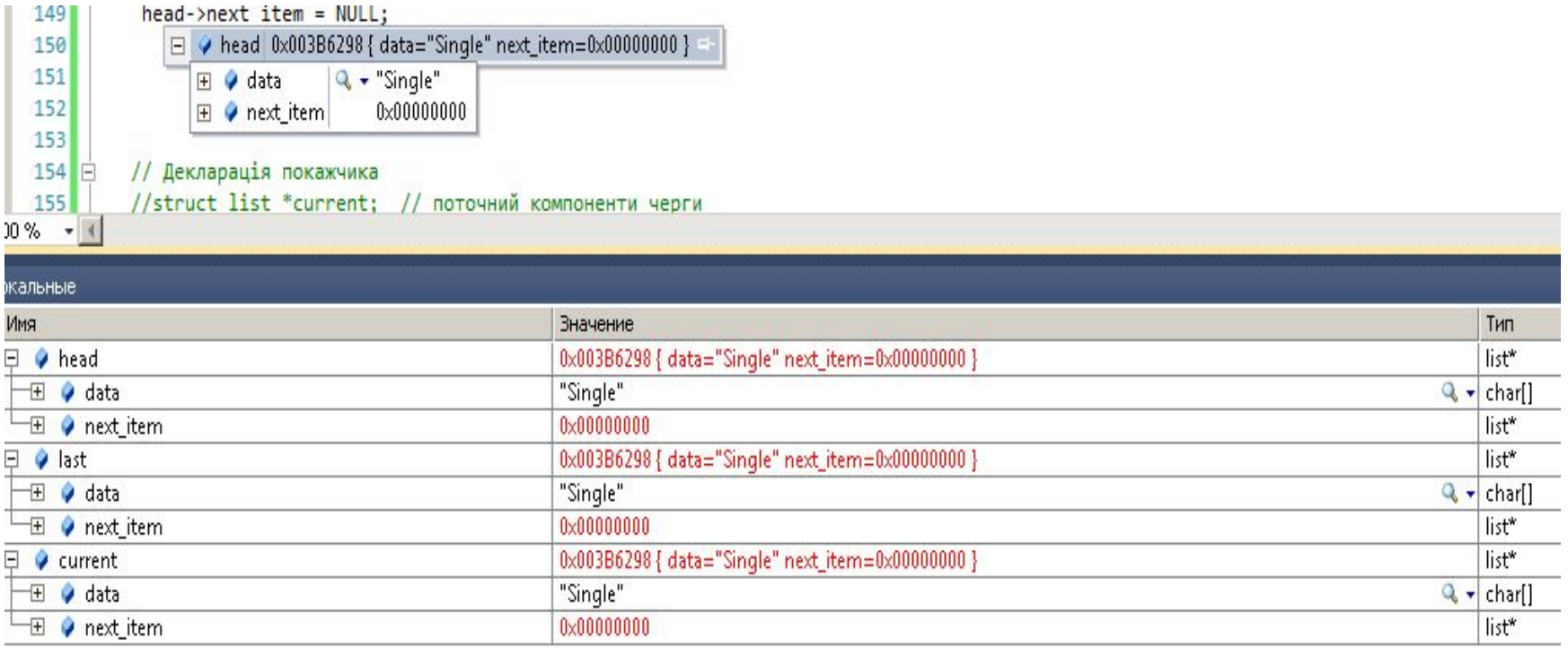


4) Ініціалізувати кінець списку

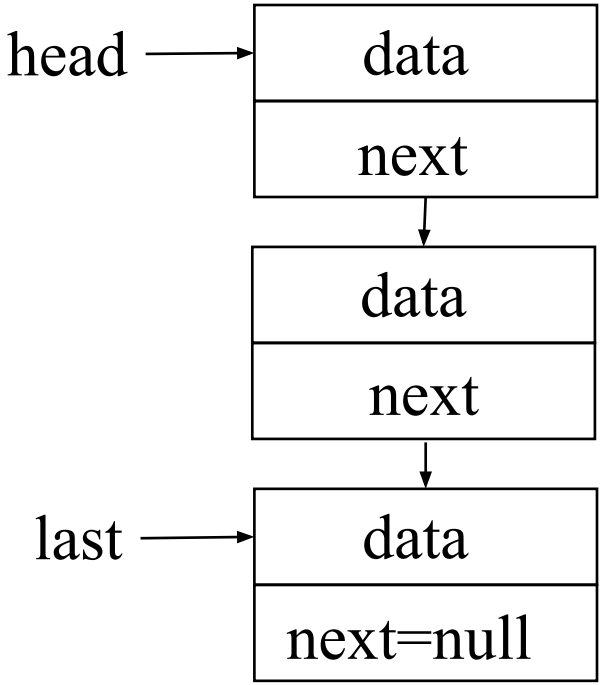
last = current;

5) Записати ознаку того, що перший елемент є останнім

head->next_item = NULL;

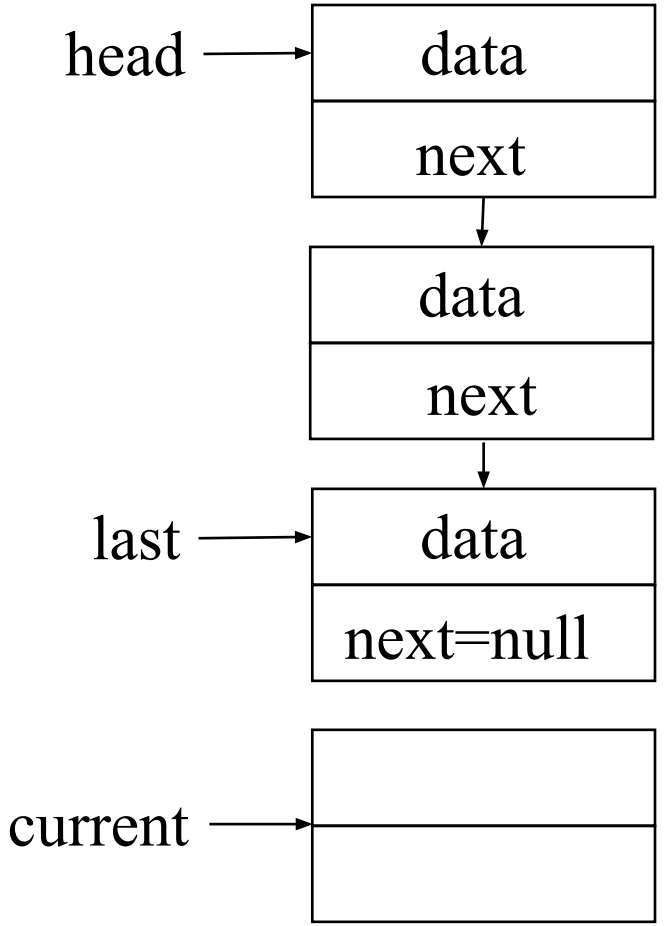


Алгоритм вставки елемента до черги



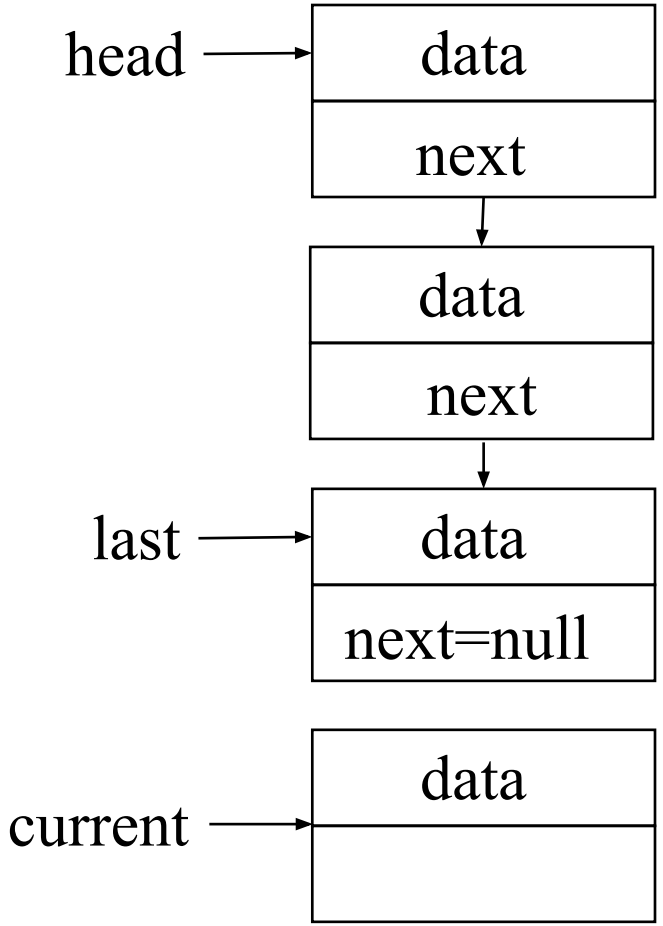
Алгоритм вставки елемента до черги

1. Виділити пам'ять для нового елемента черги



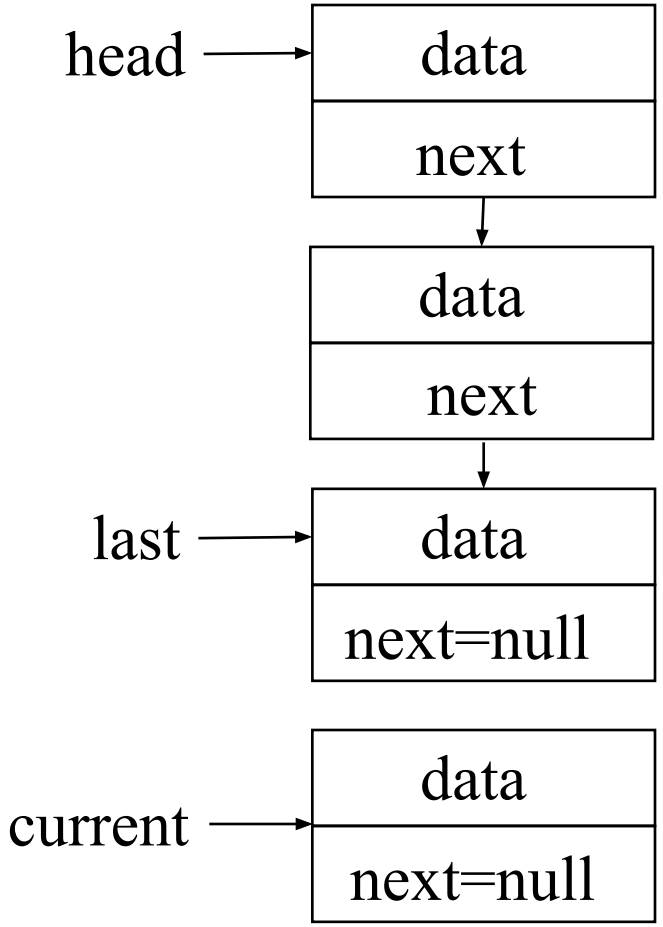
Алгоритм вставки елемента до черги

1. Виділити пам'ять для нового елемента черги
2. Ввести дані до нового елемента



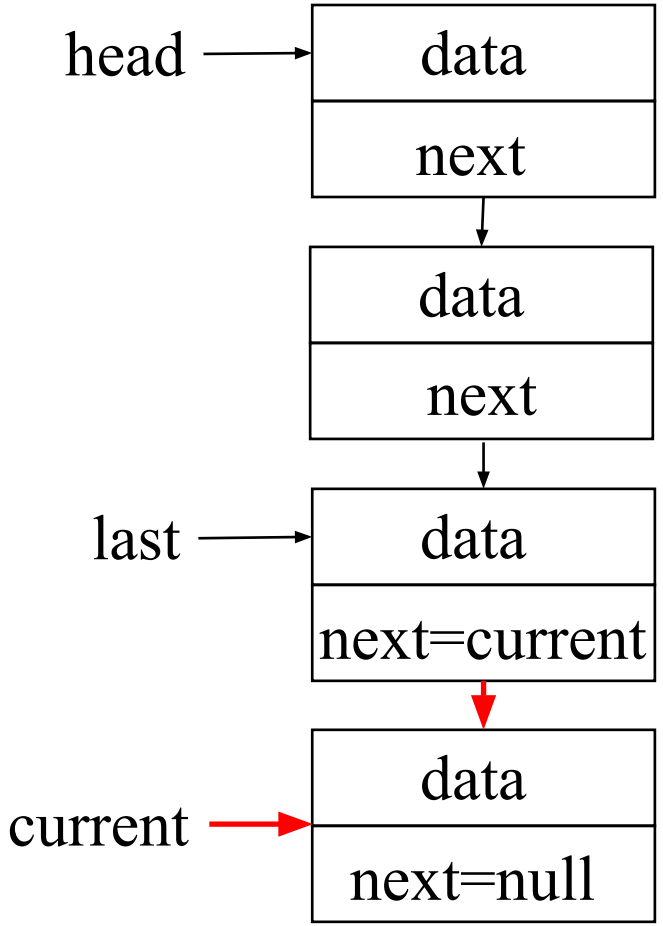
Алгоритм вставки елемента до черги

1. Виділити пам'ять для нового елемента черги
2. Ввести дані до нового елемента
3. Вважати новий елемент останнім



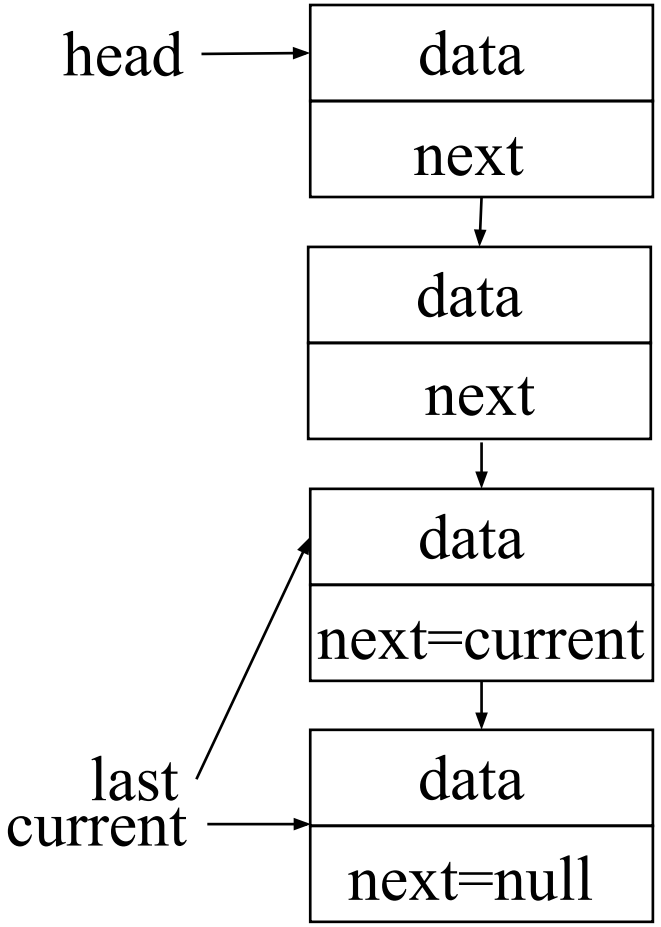
Алгоритм вставки елемента до черги

1. Виділити пам'ять для нового елемента черги
2. Ввести дані до нового елемента
3. Вважати новий елемент останнім
4. Зв'язати останній елемент черги із новоутвореним



Алгоритм вставки елемента до черги

1. Виділити пам'ять для нового елемента черги
2. Ввести дані до нового елемента
3. Вважати новий елемент останнім
4. Зв'язати останній елемент черги із новоутвореним
5. Переставити покажчик на останній елемент у черзі на новий елемент



Елементи з черги видаляються за тим самим алгоритмом, що і зі стеку.

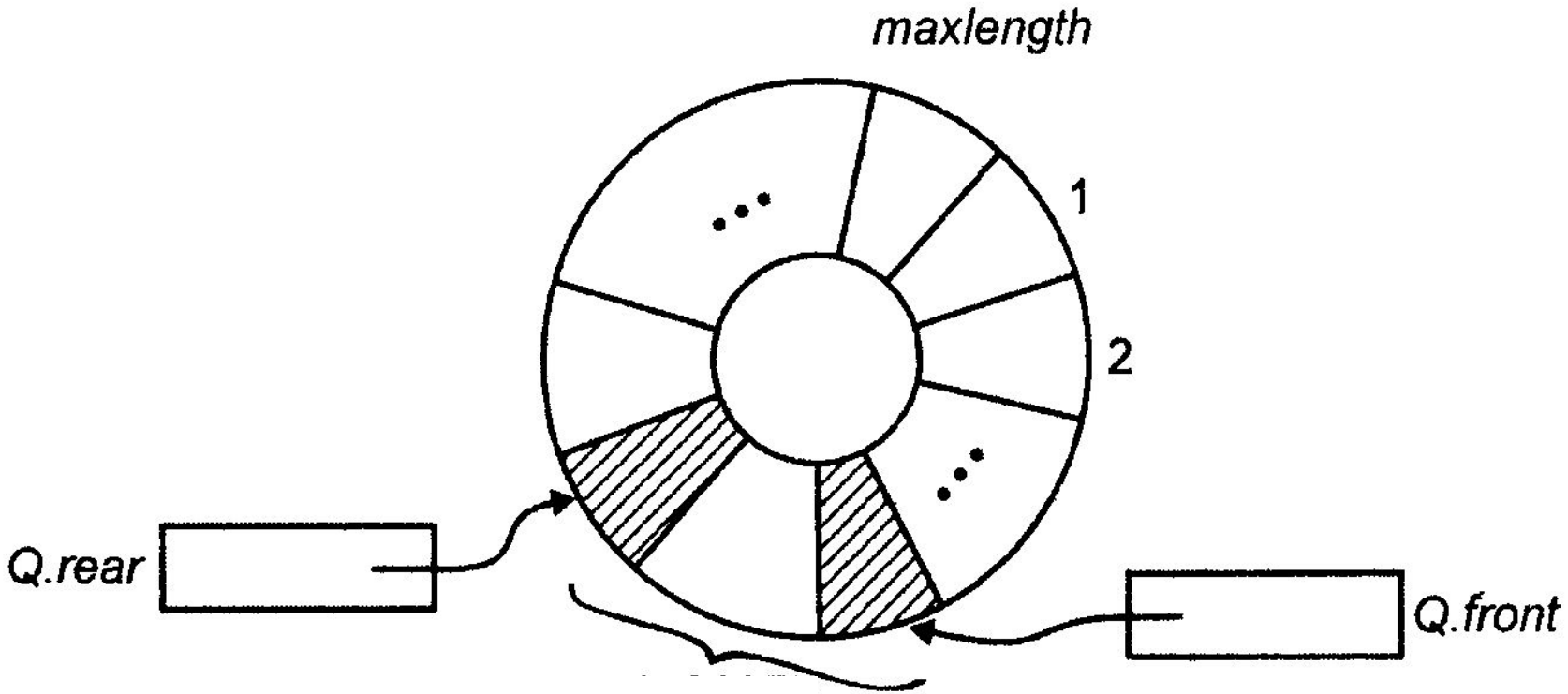
Друк елементів стеку або черги

```
void print_list(struct list *p_head)
{ struct list *current;  // поточний компонент списку
   int i = 0;
   current = p_head;
   cout <<'\n';
   while (current!= NULL)
   { cout << "I:" <<  i  <<  "  "  << current->data  <<  " \n";
      current =current->next_item;
      i++;
   }
   return;
}
```

Реалізація черг за допомогою циклічних масивів

Реалізацію списків за допомогою масивів, яка розглядалася раніше, можна застосувати для черг, але в даному випадку це не раціонально, бо за допомогою покажчика на останній елемент черги можна виконати додавання елемента за фіксоване число кроків (незалежне від довжини черги), але оператор видалення елемента, який видаляє перший елемент, вимагає переміщення всіх елементів черги на одну позицію в масиві.

Щоб уникнути цих обчислювальних витрат, представимо масив у вигляді циклічної структури, де перший запис масиву слідує за останнім



При такому представленні черги оператори додавання і видалення елемента виконуються за фіксований час, незалежний від довжини черги.

Елементи черги розташовуються в "колі" записів в послідовних позиціях, кінець черги знаходиться за годинниковою стрілкою на певній відстані від початку. Тепер для вставки нового елементу в чергу достатньо перемістити покажчик на кінець черги на одну позицію за годинниковою стрілкою і записати елемент в цю позицію. При видаленні елементу з черги треба просто перемістити покажчик на початок черги за годинниковою стрілкою на одну позицію.

Приклади, що ілюструють реалізації АТД “Черга”:

– реалізація за допомогою покажчиків (с.316-319 [4])
– ще одна реалізація за допомогою покажчиків (с.62-63 [1])
– реалізація за допомогою масивів (с.63-66 [1]).

3.5. Однозв'язний лінійний список

Література для самостійного читання:

с. 60-66 [1], с. 319-325 [4]

Стек і черга є лінійними списками, множина допустимих операцій над якими обмежена операціями над першим або останнім елементом. Розглянемо списки, над якими припустимі довільні дії.

Найбільш ефективно у спискових структурах реалізуються операції вставки та видалення елементів, оскільки вони, на відміну від операцій видалення та вставки елементів масиву, не потребують зсуву групи елементів.

Всі можливі варіанти застосування операцій вставки та видалення елементів у списку:

- створення списку, тобто внесення першого елемента до списку;
- додавання елемента в кінець списку;
- додавання елемента на початок списку;
- вставка елемента в середину списку;
- видалення елемента з початку списку;
- видалення елемента з кінця списку;
- видалення елемента з середини списку.

У загальному випадку для роботи з однозв'язним лінійним списком потрібні такі покажчики:

`head` на початок списку;

`current` на поточний елемент списку;

`previous` на елемент, розташований перед поточним;

`newptr` на елемент, що додається до списку;

`last` на кінець списку.

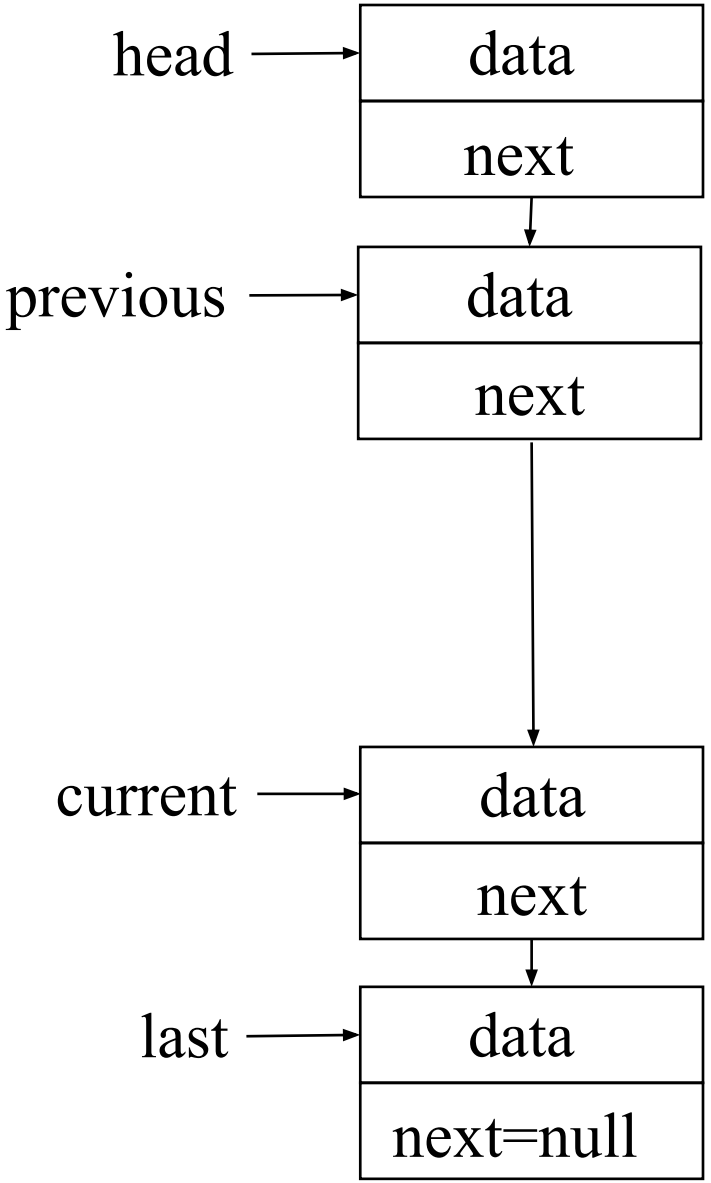
У різних задачах можуть використовуватися не всі покажчики.

Додавання елемента в кінець списку виконується за алгоритмом додавання елемента до черги, а на початок списку — за алгоритмом додавання елемента до стеку.

Операція видалення елемента з початку списку здійснюється за алгоритмом видалення елемента зі стеку або з черги.

Алгоритм вставки елемента в середину списку

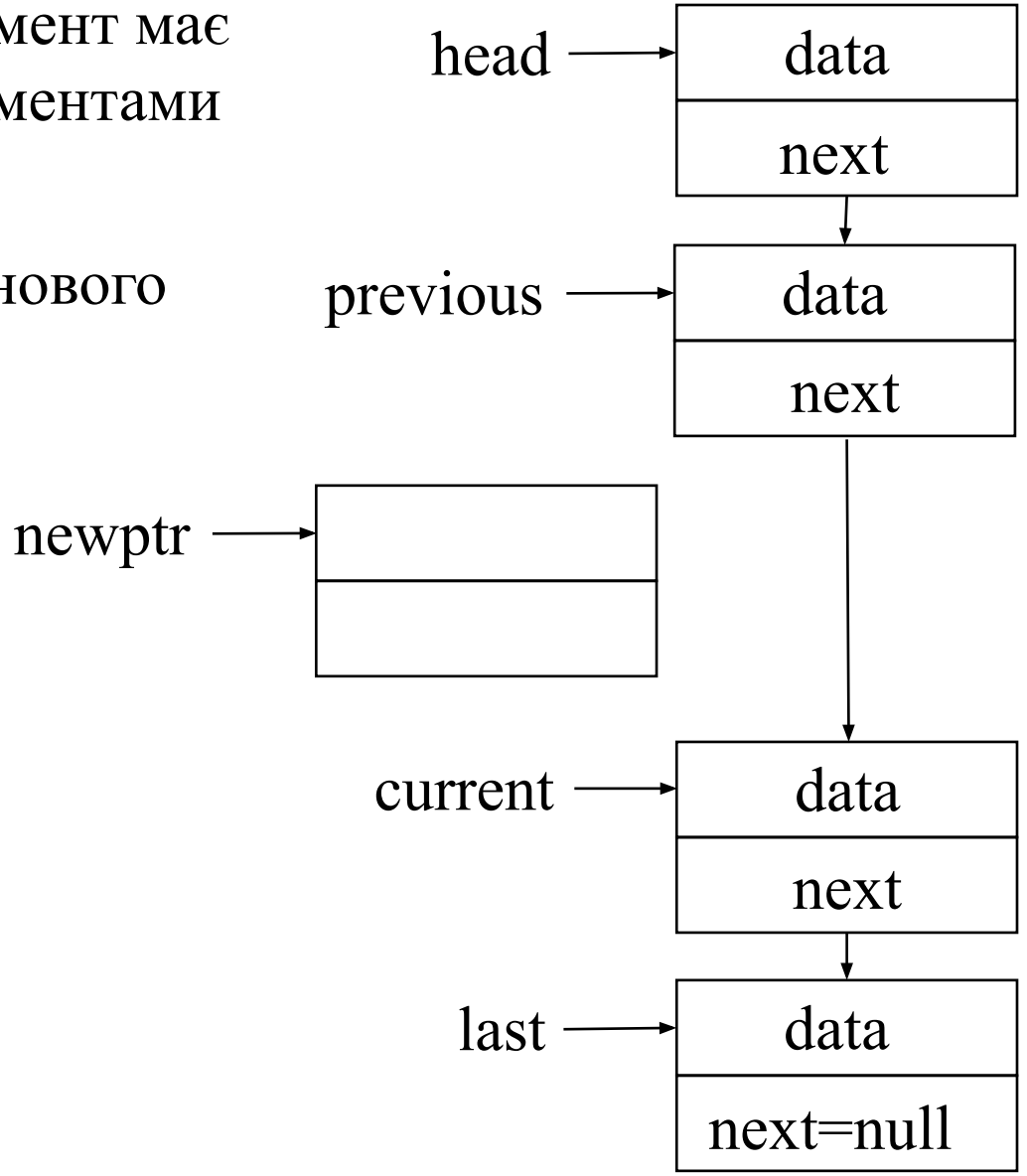
Вважаємо, що новий елемент має бути вставлений між елементами `previous` і `current`.



Алгоритм вставки елемента в середину списку

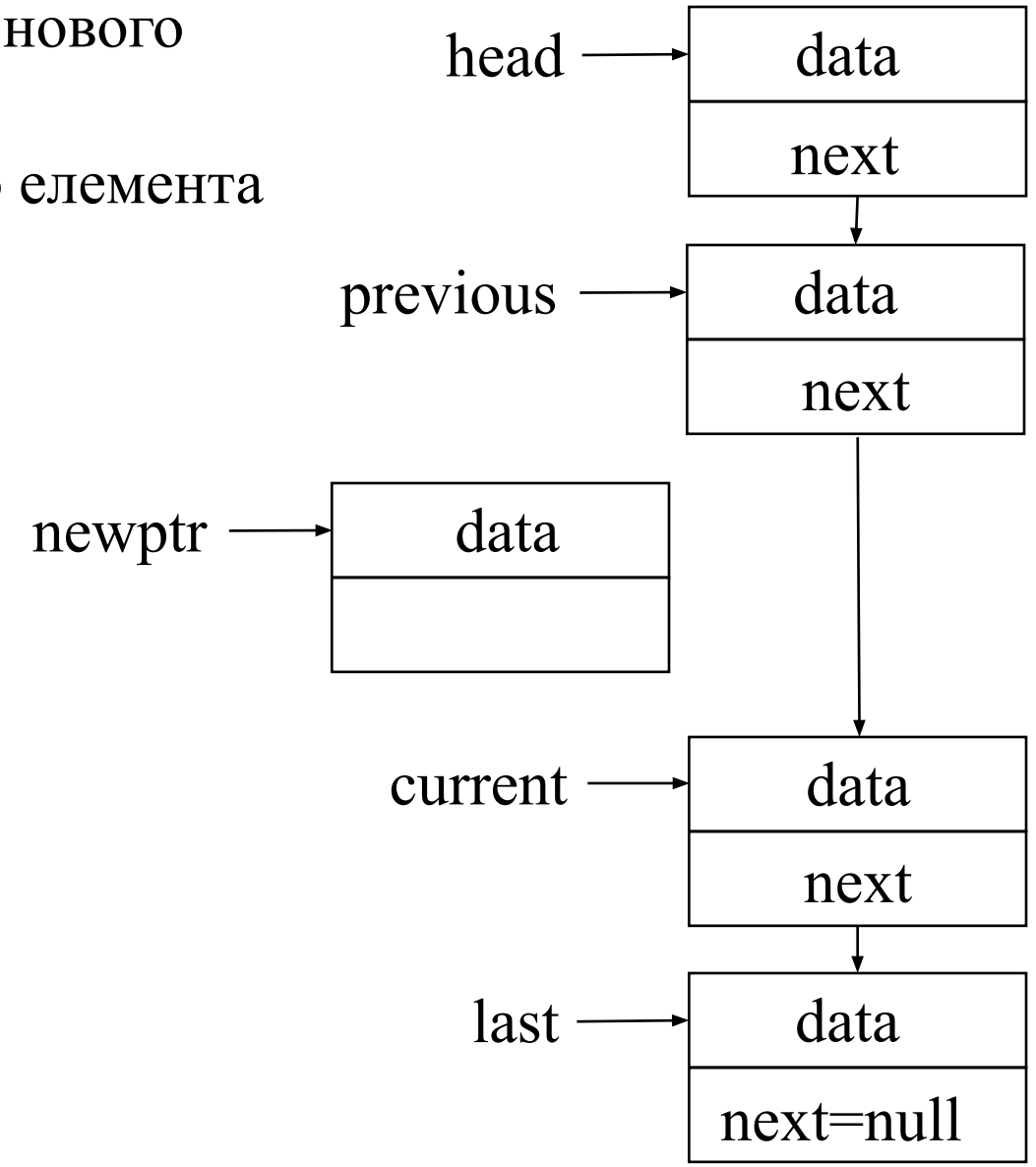
Вважаємо, що новий елемент має бути вставлений між елементами `previous` і `current`.

1. Виділити пам'ять для нового елемента



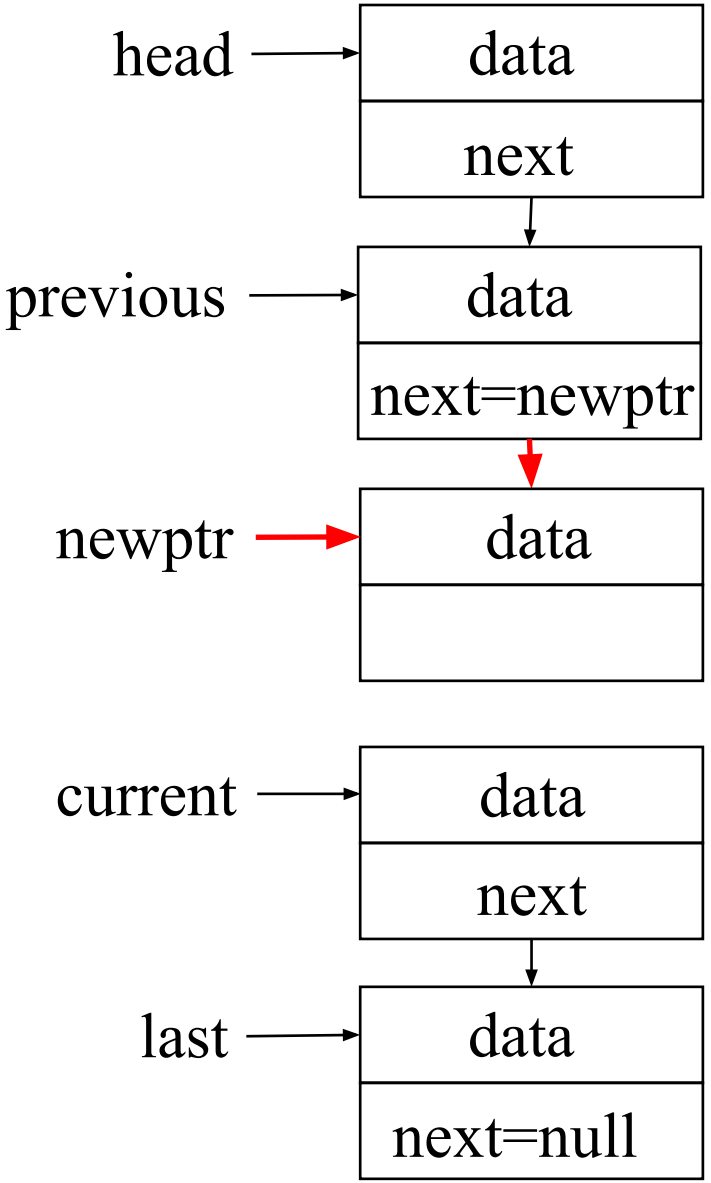
Алгоритм вставки елемента до черги

1. Виділити пам'ять для нового елемента
2. Ввести дані до нового елемента



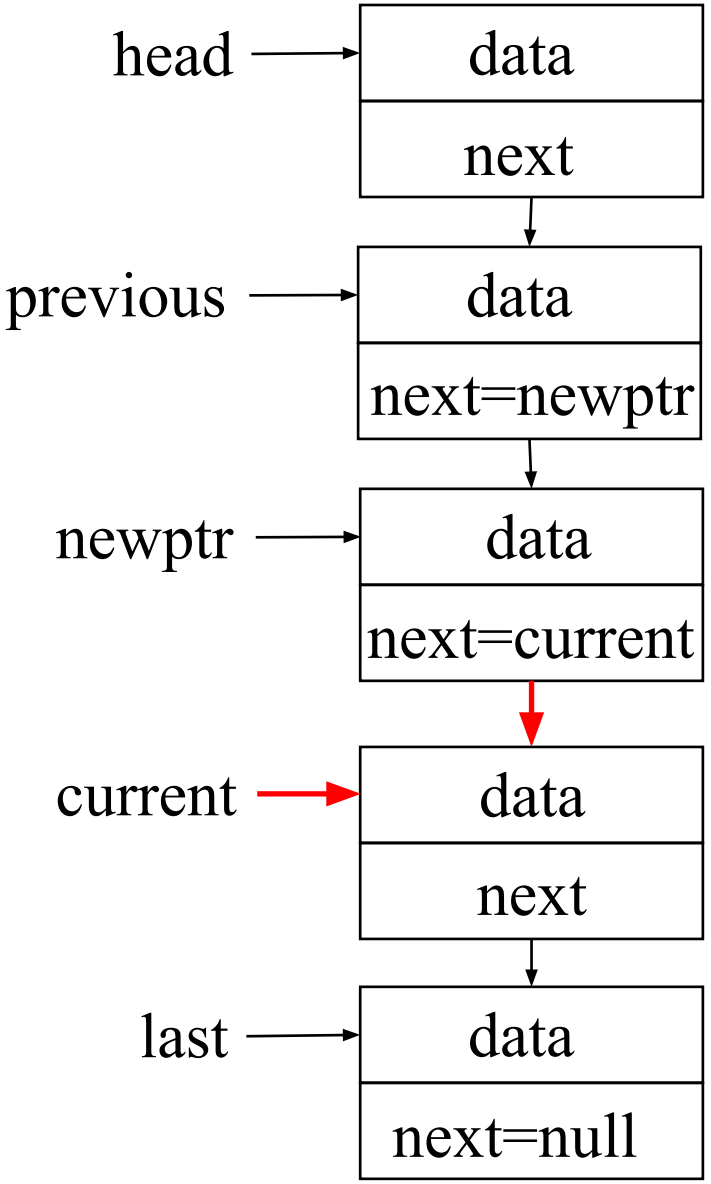
Алгоритм вставки елемента в середину списку

1. Виділити пам'ять для нового елемента
2. Ввести дані до нового елемента
3. Новий елемент вважати наступним для `previous`



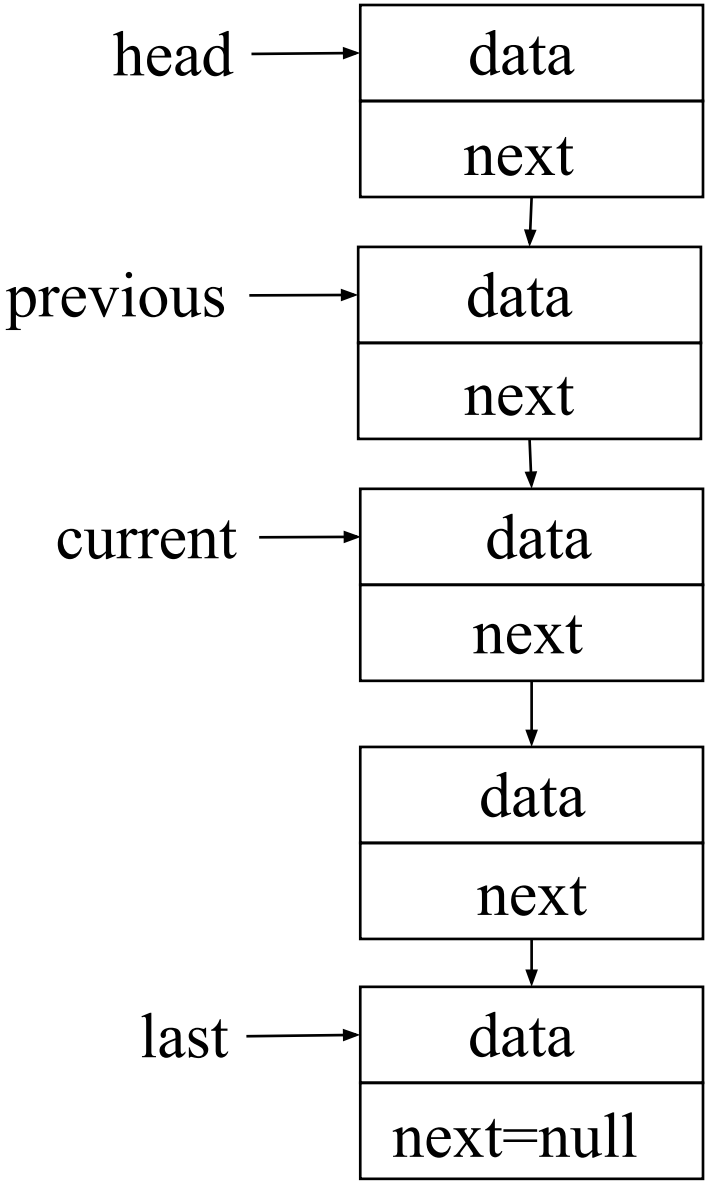
Алгоритм вставки елемента в середину списку

1. Виділити пам'ять для нового елемента
2. Ввести дані до нового елемента
3. Новий елемент вважати наступним для `previous`
4. Для нового елемента вважати наступним `current`



Алгоритм видалення елемента з середини списку

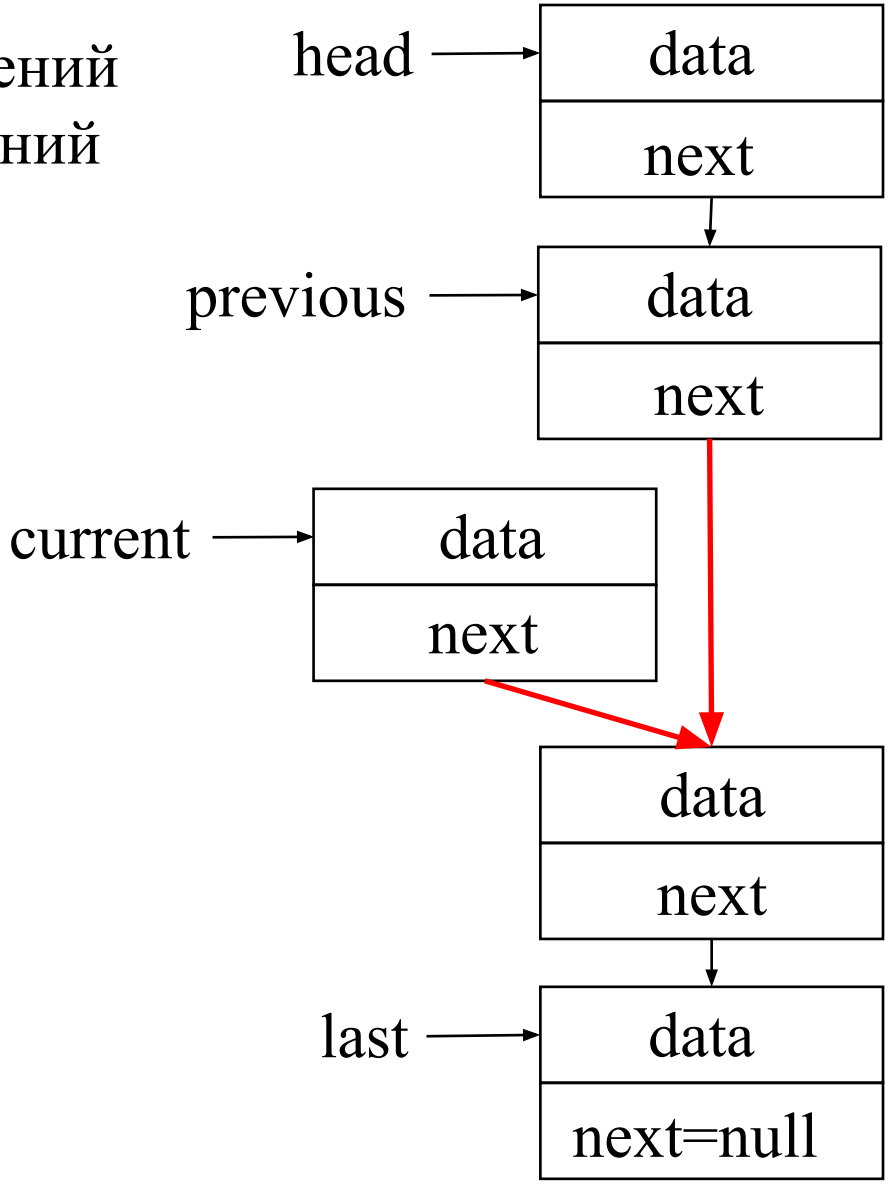
Вважаємо, що має бути видалений елемент `current`, розташований безпосередньо за елементом `previous`.



Алгоритм видалення елемента з середини списку

Вважаємо, що має бути видалений елемент `current`, розташований безпосередньо за елементом `previous` .

1. Вважати, що за елементом `previous` буде розташований той елемент, що раніше знаходився за елементом `current`

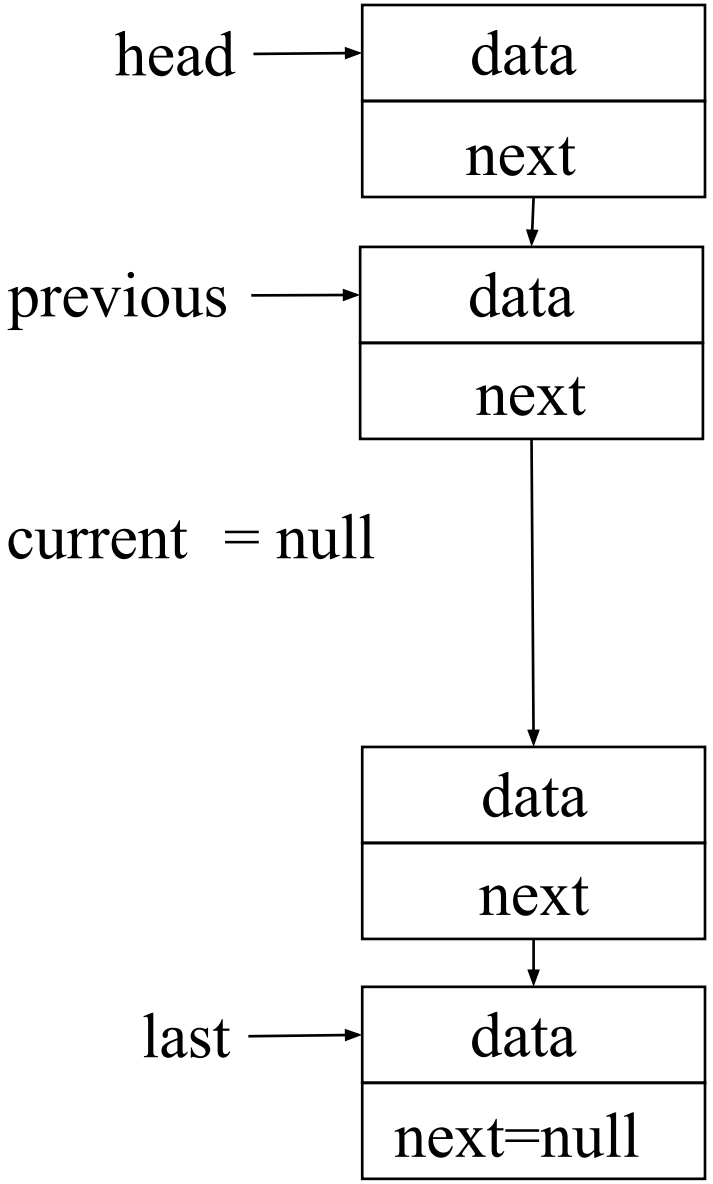


Алгоритм видалення елемента з середини списку

Вважаємо, що має бути видалений елемент `current`, розташований безпосередньо за елементом `previous` .

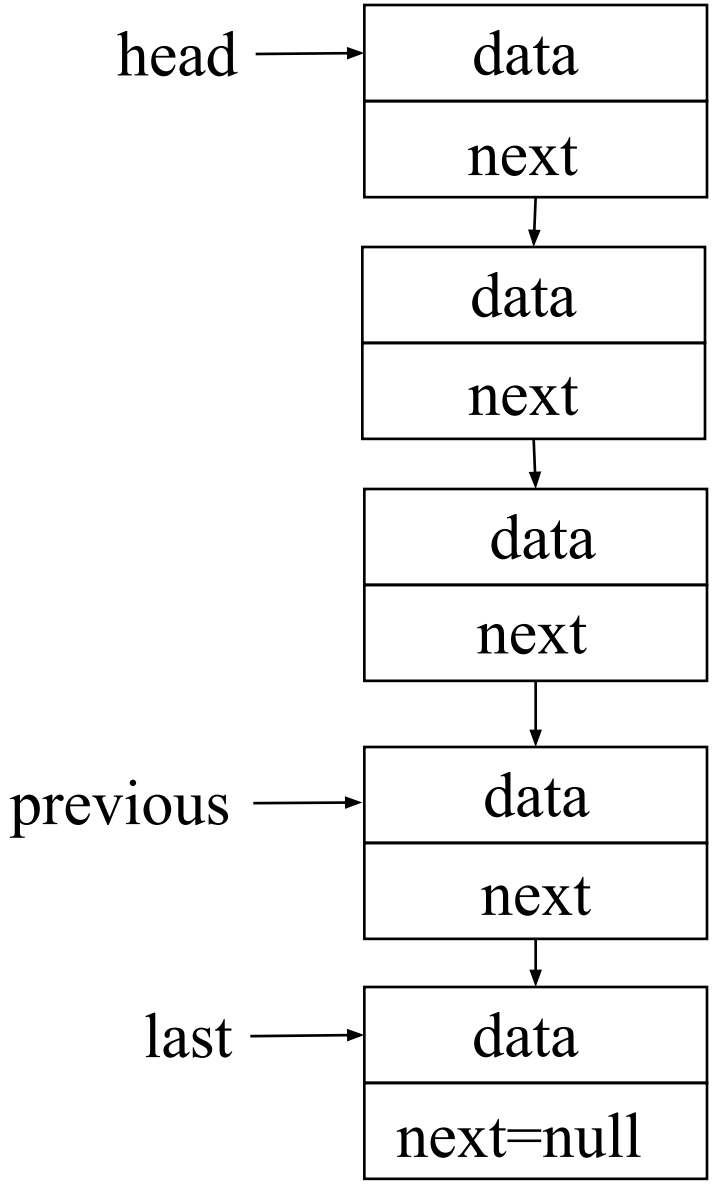
1. Вважати, що за елементом `previous` буде розташований той елемент, що раніше знаходився за елементом `current`

2. Звільнити пам'ять із-під елемента `current`



Алгоритм видалення елемента з кінця списку

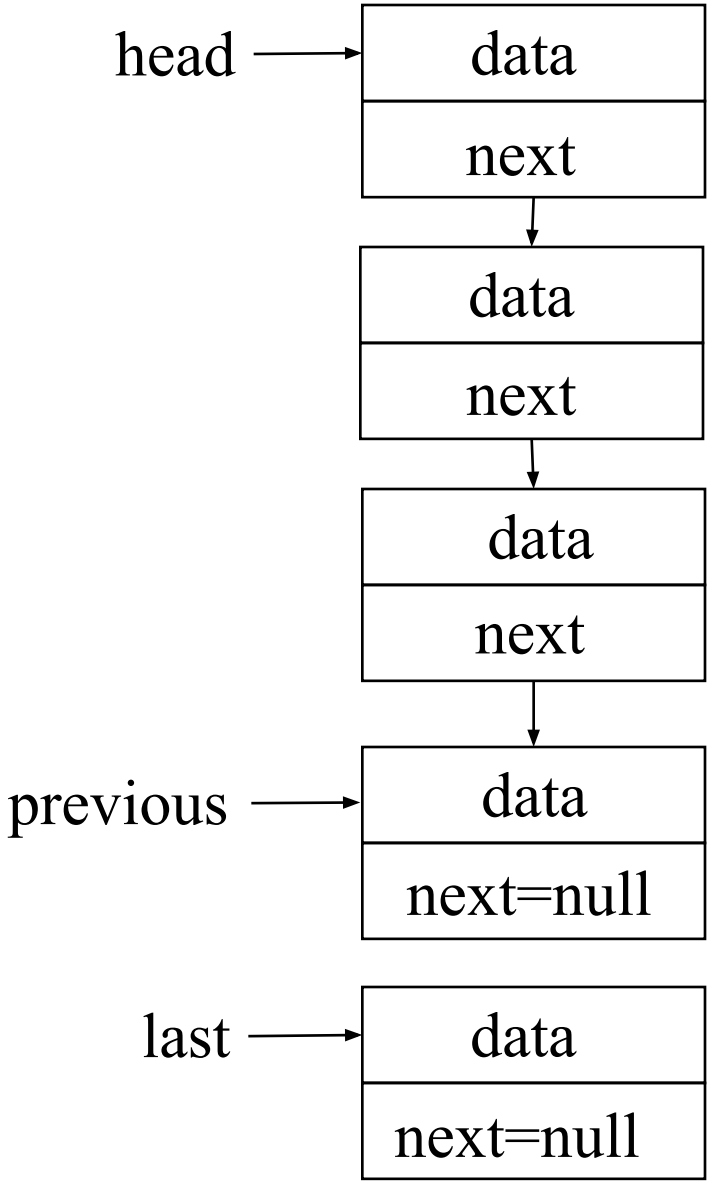
Вважаємо, що на передостанній елемент посилається покажчик `previous`.



Алгоритм видалення елемента з кінця списку

Вважаємо, що на передостанній елемент посилається покажчик `previous`.

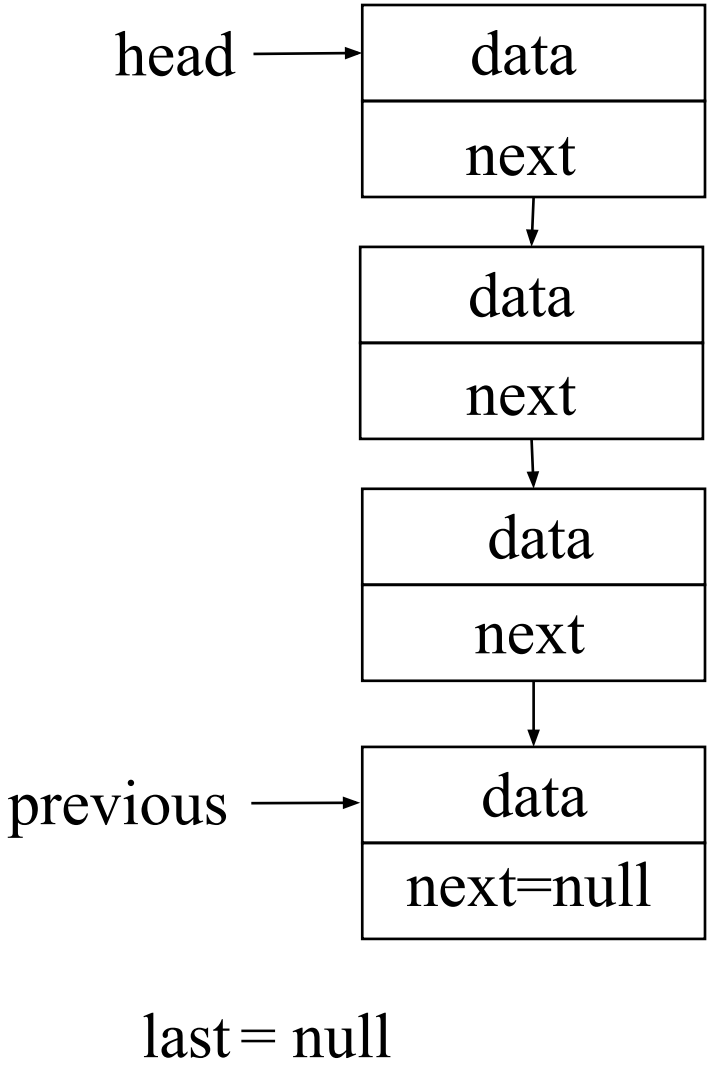
1. Записати до передостаннього елемента ознаку кінця списку



Алгоритм видалення елемента з кінця списку

Вважаємо, що на передостанній елемент посилається покажчик `previous`.

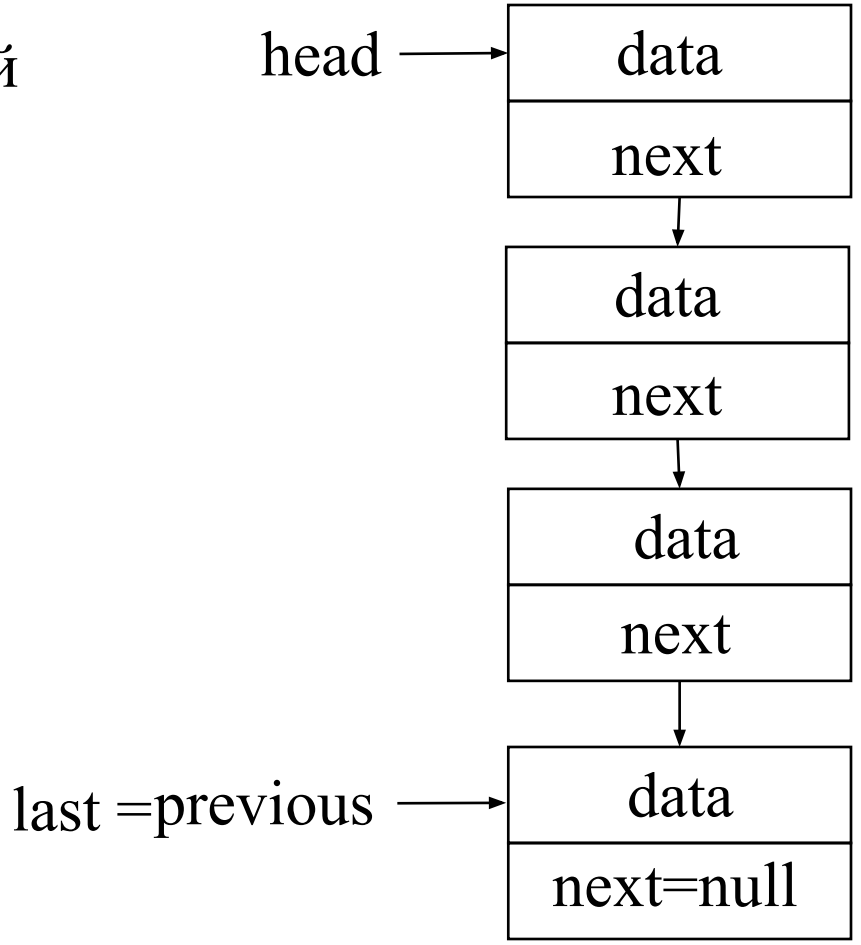
1. Записати до передостаннього елемента ознаку кінця списку
2. Звільнити пам'ять із-під колишнього останнього елемента



Алгоритм видалення елемента з кінця списку

Вважаємо, що на передостанній елемент посилається покажчик `previous`.

1. Записати до передостаннього елемента ознаку кінця списку
2. Звільнити пам'ять із-під колишнього останнього елемента
3. Вважати останнім колишній передостанній елемент



Приклад. Алгоритм роботи з алфавітним переліком слів.

1. Вважати список порожнім.
2. Вивести меню для роботи зі списком.
3. Якщо натиснута клавіша I, додати елемент до списку.
   3.1. Виділити пам'ять для нового елемента.
   3.2. Ввести нове слово та ініціалізувати ним поле даних нового елемента.
   3.3. Якщо список порожній, вважати щойно утворений елемент списком.
   3.4. Якщо список непорожній, визначити місце розташування нового елемента та вставити його до списку.

4. Якщо натиснута клавіша D, видалити елемент зі списку.
   4.1. Ввести слово, що видаляється.
   4.2. Якщо список порожній, вивести відповідне повідомлення.
   4.3. Якщо список непорожній, проглядати значення елементів списку доти, доки введене слово не буде знайдено або доки список не буде вичерпано.
   4.4. Якщо елемент із введеним значенням поля даних було знайдено, то його слід видалити.
   4.5. Якщо введене слово не збігається зі значенням інформаційного поля жодного елемента списку, вивести повідомлення про відсутність шуканого елемента у списку.
5. Якщо натиснута клавіша Q, вийти з програми.

Приклади, що ілюструють реалізації АТД “Однозв'язний лінійний список”:

– реалізація за допомогою покажчиків (с.319-325 [4])
– ще одна реалізація за допомогою покажчиків (с.50-53 [1])
– реалізація за допомогою масивів (с.48-50 [1]).

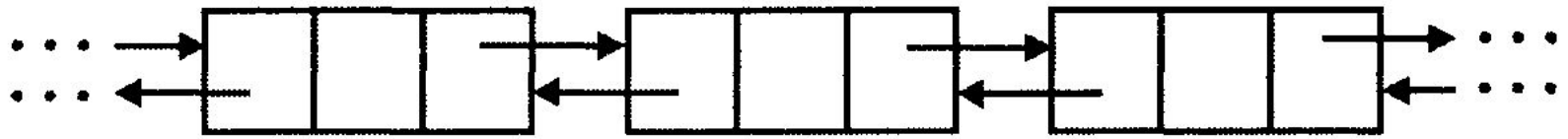
3.6. Двозв'язний лінійний список

Література для самостійного читання:

с. 57-58 [1]

Часто виникає необхідність організувати ефективне пересування по списку як в прямому, так і в зворотному напрямах. Або по заданому елементу потрібно швидко знайти передуючий йому і наступний елементи. У цих ситуаціях можна дати кожному запису покажчики і на наступний, і на попередній записи у списку, тобто організувати двічі зв'язний список.

З прикладом реалізації можна ознайомитись в (с.57-58 [1]).



3.7. Відображення

Література для самостійного читання:

с. 66-68 [1]

Відображення - це функція, визначена на множині елементів одного типу (області визначення), що приймає значення з множини елементів другого типу (області значень) (звичайно, типи можуть співпадати).

Той факт, що відображення M ставить у відповідність елемент d з області визначення елементу r з області значень, записуватимемо як $M(d)=r$. Деякі відображення, подібні $\text{square}(i)=i^2$, легко реалізувати з допомогою функцій і арифметичних виразів мови програмування. Але для багатьох відображень немає очевидних способів реалізації, окрім зберігання для кожного d значення $M(d)$. Наприклад, для реалізації функції, що ставить у відповідність працівникам їх зарплату, потрібно зберігати поточний заробіток кожного працівника.

Перелік операторів, які можна виконати над відображенням M.

– перетворення відображення на порожнє;
– призначення $M(d)=r$ незалежно від того, як $M(d)$ було визначено раніше;
– повернення значення $M(d)$, якщо воно визначено, і повідомлення про невизначеність в протилежному випадку.

Реалізація відображень за допомогою масивів

У багатьох випадках тип елементів області визначення відображення є простим типом, який можна використовувати як тип індексів масивів.

Такі відображення просто реалізувати за допомогою масивів, припускаючи, що деякі значення з області значень можуть мати статус "невизначений". Наприклад, для відображення crypt, описаного вище, область значень можна визначити інакше, ніж 'A'...'Z', і використовувати символ '?' для позначення "невизначений".

Реалізація відображень за допомогою списків

Існує багато реалізацій відображень з кінцевою областю визначення. Наприклад, в багатьох ситуаціях відмінним вибором будуть хеш-таблиці. Інші відображення з кінцевою областю визначення можна представити у вигляді списку пар (d_1, r_1) (d_2, r_2) .... (d_k, r_k), де d_1 d_2 ..., d_k - всі поточні елементи області визначення, а r_1, r_2 ..., r_k - значення, що асоціюються з d_i (i = 1, 2 ..., k). Далі можна використовувати будь-яку реалізацію списків.

Приклади, що ілюструють реалізації АТД “Відображення”:

- реалізація за допомогою покажчиків (с.68 [1])
- реалізація за допомогою масивів (с.67 [1])
- реалізація за допомогою хеш-таблиць (с.116-128 [1]).

3.8. АТД “Дерево”

Література для самостійного читання:

с. 77-89 [1], с. 326-330 [4]

Розглянуті раніше списки, стеки та черги належать до лінійних динамічних структур даних. Визначальною характеристикою лінійних структур є те, що зв'язок між їхніми компонентами описується в термінах «попередній-наступний», тобто для кожного компонента лінійної структури визначено не більше одного попереднього та не більше одного наступного компонента.

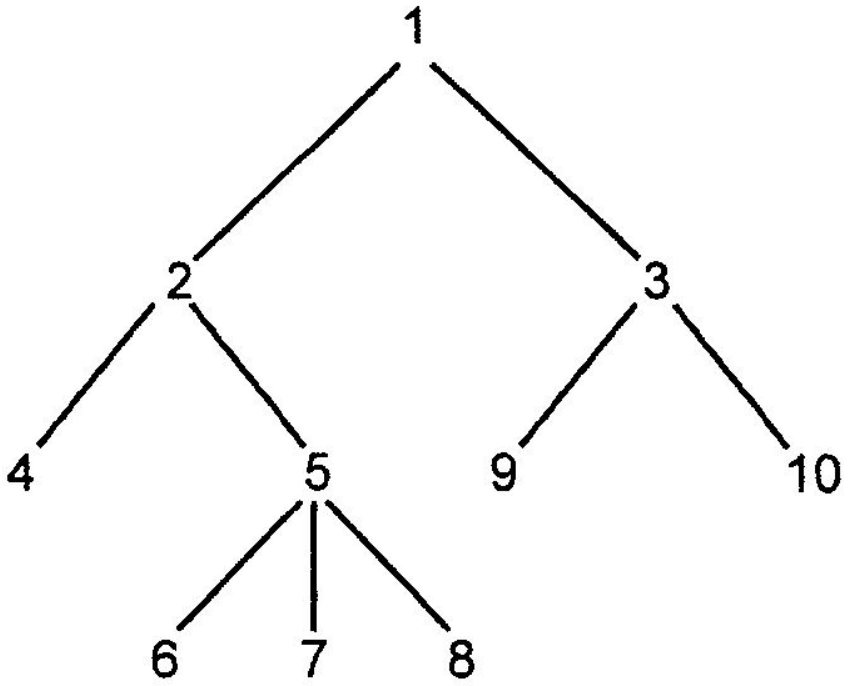
Деревоподібна структура даних, або дерево, є нелінійною структурою, що зображує ієрархічні зв'язки типу «предок-нащадок»: компонент-предок може мати багато нащадків, хоча для кожного компонента-нащадка визначено не більше одного предка.

Щоб згадати термінологію можна почитати (с.77-83 [1], с.326-327 [4]).

Представлення дерев за допомогою масивів

Нехай T - дерево з вузлами 1, 2 ..., n. Найпростішим представленням дерева T, що підтримує оператор визначення батька вузла, буде лінійний масив A, де кожен елемент $A[i]$ є покажчиком або курсором на батька вузла i. Корінь дерева T відрізняється від інших вузлів тим, що має нульовий покажчик або покажчик на самого себе як на батька.

Дане уявлення використовує властивість дерев, що кожен вузол, відмінний від кореня, має тільки одного батька. Використовуючи це уявлення, батька будь-якого вузла можна знайти за фіксований час. Проходження по будь-якому шляху, тобто перехід по вузлах від батька до батька, можна виконати за час, пропорційний кількості вузлів шляху.



дерево

	1	2	3	4	5	6	7	8	9	10
A	0	1	1	2	2	5	5	5	3	3

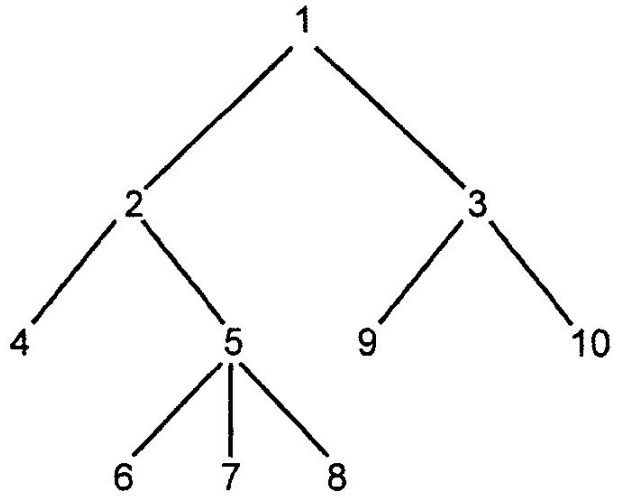
курсори на батьків

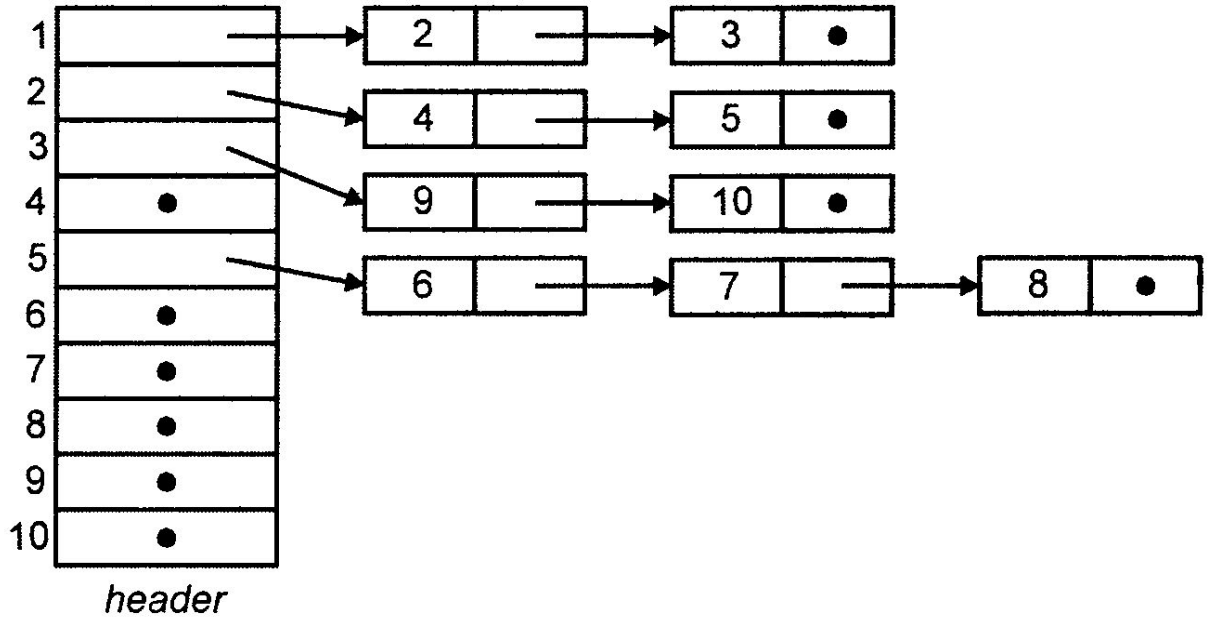
Використання покажчиків або курсорів на батьків не допомагає в реалізації операторів, що вимагають інформацію про синів. Використовуючи описане представлення, вкрай важко для даного вузла *n* знайти його синів. Крім того, в цьому випадку неможливо визначити порядок синів вузла (тобто який син знаходиться правішим або лівішим за іншого сина).

З прикладом реалізації можна ознайомитись в (с.86 [1]).

Представлення дерев з використанням списків синів

Важливий і корисний спосіб представлення дерев полягає у формуванні для кожного вузла списку його синів. Ці списки можна представити будь-яким методом для представлення списків, але, оскільки число синів у різних вузлів може бути різним, найчастіше для цих цілей застосовуються зв'язані списки.

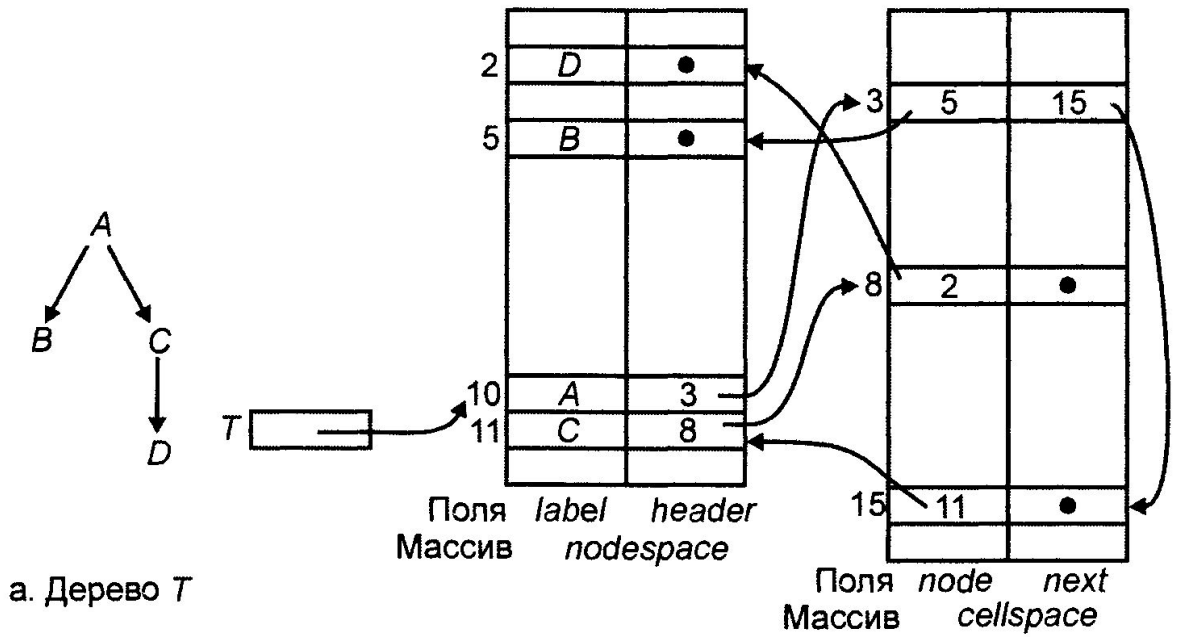




З прикладом реалізації можна ознайомитись в (с.87-88 [1]).

Серед недоліків такої структури даних можна назвати те, що вона не дозволяє створювати великі дерева з малих. Це є наслідком того, що всі дерева спільно використовують один масив для представлення зв'язаних списків синів; по суті, кожне дерево має власний масив заголовків для своїх вузлів.

З прикладом реалізації, що виправляє цей недолік, можна ознайомитись в (с.88-91 [1]).



а. Дерево *T*

б. структури даних

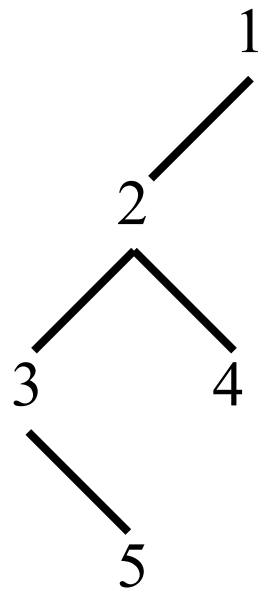
3.9. Бінарні дерева

Література для самостійного читання:

с. 91-99 [1], с. 328-336 [4]

Представлення бінарних дерев за допомогою масивів

Якщо іменами вузлів бінарного дерева є їх номери 1,2, … *n*, то підходящою структурою для представлення цього дерева може бути масив записів з полями “лівий син” та “правий син”.



номер вузла	поле лівий син	поле правий син
1	2	0
2	3	4
3	0	5
4	0	0
5	0	0

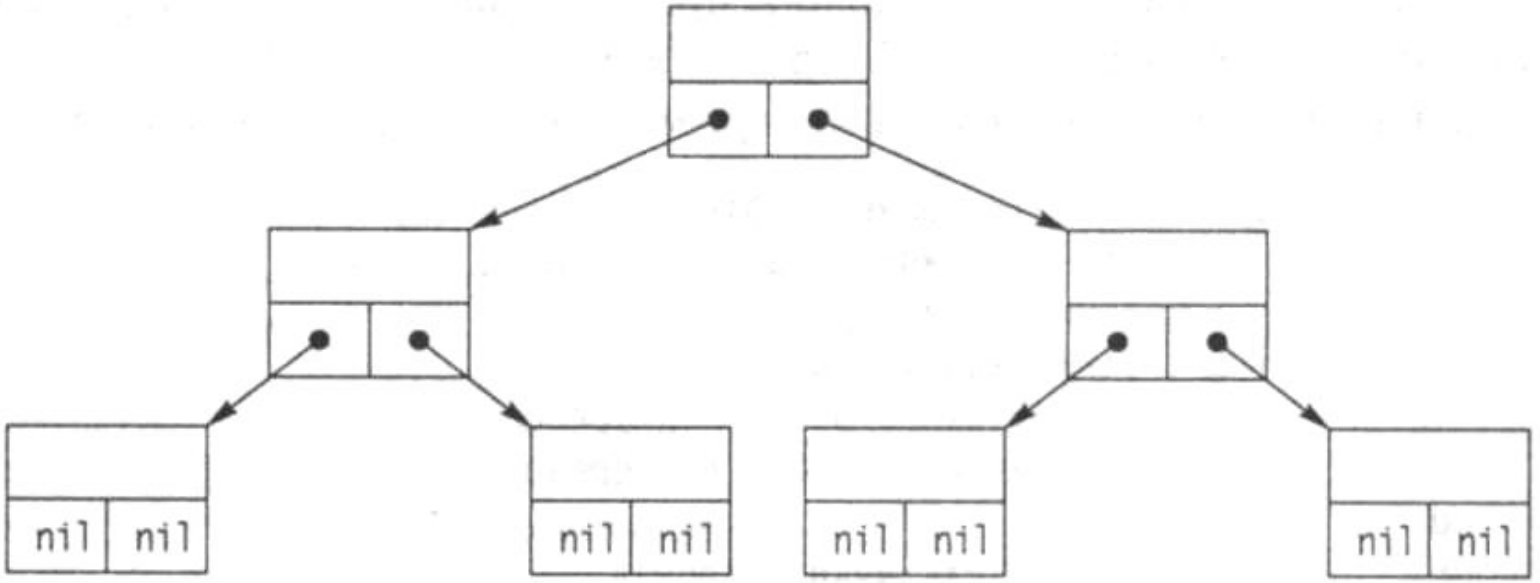
Представлення бінарних дерев за допомогою нелінійних динамічних структур

Будь-який вузол бінарного дерева може бути зв'язаним не більше ніж із двома піддеревами, що називаються *лівим* і *правим* піддеревами вузла. Оголошення типу вузла бінарного дерева на мові Pascal може бути таким:

```
type ptr=^Node;              {тип покажчика на вузол дерева}
     Node=record             {тип вузла дерева              }
            data:string;     {інформаційне поле вузла       }
            left,right:ptr;  {покажчики на ліве та
                              праве піддерево               }
          end;
```

Для листків покажчики left та right мають значення nil

Приклад бінарного дерева як динамічної структури даних



Алгоритми роботи з бінарними деревами

Створення бінарного дерева

Найпростіший спосіб побудови бінарного дерева полягає у створенні дерева симетричної структури із наперед відомою кількістю вузлів. Усі вузли-нащадки, що створюються, рівномірно розподіляються зліва та справа від кожного вузла-предка. При цьому досягається мінімально можлива глибина для заданої кількості вузлів дерева. Правило рівномірного розподілу n вузлів можна визначити рекурсивно:

- перший вузол вважати коренем дерева;
- створити ліве піддерево з кількістю вузлів $nleft=n$ div 2;
- створити праве піддерево з кількістю вузлів $nright=n-nleft-1$.

Приклад. Створення бінарного дерева із заданою користувачем кількістю вузлів.

Оскільки структура дерева визначена рекурсивно, то для його створення та відображення можна розробити рекурсивні підпрограми.

```
program ex10_6; {створення та відображення бінарного дерева}
type ptr=^node;              {тип покажчика на вузол дерева     }
     node=record             {тип вузла дерева                  }
            data:string;     {інформаційне поле вузла           }
            left,            {покажчик на ліве  піддерево       }
            right:ptr;       {покажчик на праве піддерево       }
          end;
var n:integer;               {кількість вузлів дерева           }
    root:ptr;                {покажчик на корінь дерева         }
```

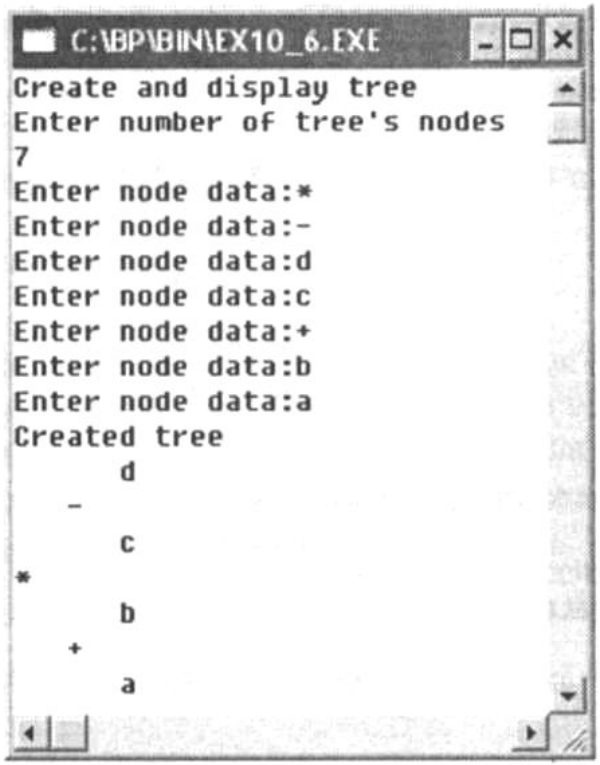
Функція створення дерева tree отримує один цілочисловий параметр AmountNode, що визначає кількість вузлів дерева та повертає покажчик на його корінь. Якщо кількість вузлів дорівнює нулю, дерево порожнє, а отже, виконано умову завершення рекурсії і функція має повернути значення nil. Якщо кількість вузлів дерева відрізняється від нуля, необхідно виділити пам'ять для кореня дерева, за наведеними вище формулами обчислити кількість вузлів у лівому та правому піддеревах і двічі рекурсивно викликати функцію tree для створення піддерев. Для посилання на корінь дерева використано локальний покажчик newnode. Значення покажчиків на корені піддерев, що їх повертатиме функція tree в результаті рекурсивних викликів, слід присвоїти полям left та right змінної newnode^.

```
{============== створення бінарного дерева ================}
function Tree(AmountNode:integer):ptr;
{AmountNode – кількість вузлів дерева,
 ptr – покажчик на корінь дерева}
var newnode:ptr;                {покажчик на новий вузол      }
    LeftNodes,                  {кількість вузлів у лівому    }
    RightNodes:integer;         {і правому піддеревах         }
    str:string;                 {інформаційне поле вузла      }
begin
  if AmountNode =0 then         {якщо вузлів немає,           }
     tree:=nil                  {дерево порожнє               }
    else
    begin
      LeftNodes:= AmountNode div 2; {кількість вузлів
                                     у піддеревах           }
      RightNodes:= AmountNode – LeftNodes -1;
      write('Enter node data:');
      readln(str);
      New(newnode);       {виділити пам'ять для нового вузла}
      with newnode^ do
      begin
        data:=str;        {задати інформаційне поле вузла   }
                          {створити ліве піддерево          }
        left:=Tree(LeftNodes);
                          {створити праве піддерево         }
        right:=Tree(RightNodes);
      end;
      Tree:=newnode;      {значення, що повертається        }
    end;
end;
```

Дерево відображатиме рекурсивна процедура printtree. Піддерево рівня *L* виводитиметься так: спочатку буде відображене ліве піддерево, потім - корінь піддерева рівня *L*, перед яким буде виведено *L* пробілів, далі - праве піддерево.

```
{================ відображення дерева ====================}
procedure Printtree(RootTree:ptr;L:integer);
{RootTree – покажчик на корінь дерева; L – номер рівня     }
var i:integer;                              {параметр циклу}
begin
   if RootTree<>nil then        {дерево не порожнє         }
      with RootTree^ do
      begin
        Printtree(left,L+1);    {вивести ліве піддерево    }
        for i:=1 to L do write('    ');
        writeln(data);          {вивести значення вузла    }
        Printtree(right,L+1);   {вивести праве піддерево   }
      end;
end;
```

```
{================== основна програма  ====================}
begin
  writeln('Create and display tree');
  writeln('Enter number of tree''s nodes');
  readln(n);              {задати кількість вузлів дерева    }
  root:=Tree(n);          {створити дерево                   }
  writeln('Created tree');
  Printtree(root,0);      {відобразити дерево                }
  readln;
end.
```

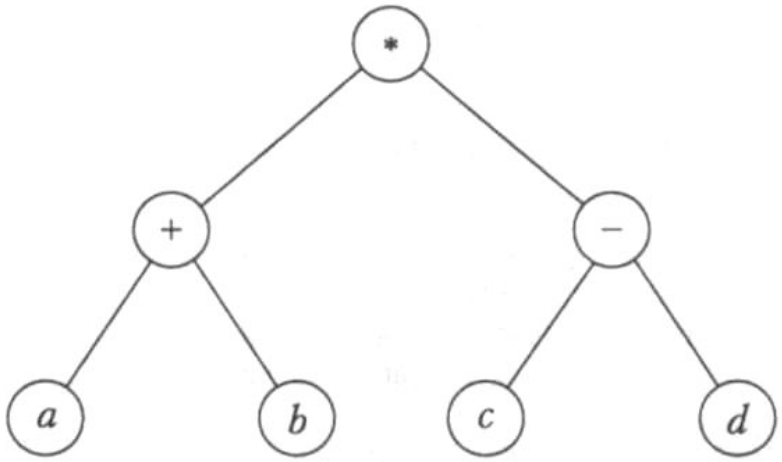


Обхід дерева

Алгоритм доступу до всіх вузлів дерева називається *обходом дерева.* Трьома основними способами обходу дерева є обхід зверху вниз (в прямому порядку), зліва направо (в симетричному порядку) та знизу вверх (в зворотньому порядку).

У результаті обходу синтаксичного дерева зверху вниз утворюється *префіксна* форма виразу, при обході знизу вверх — *постфіксна* форма, а при обході зліва направо — *інфіксна* форма.

Результати обходу дерева.



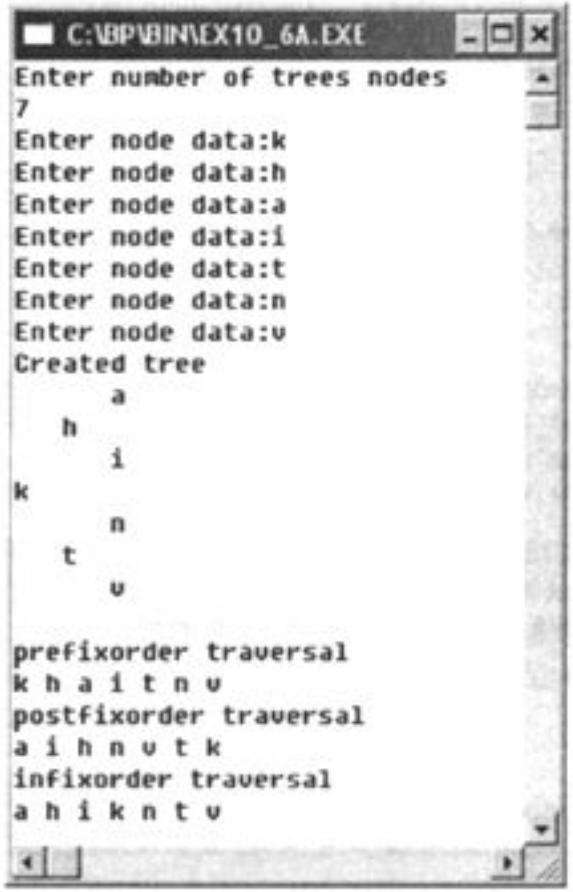
Спосіб обходу	Послідовність символів
Зверху вниз	* + ab-cd
Зліва направо	a + b* c-d
Знизу вверх	ab + cd-*

Будь-який спосіб обходу дерева можна реалізувати рекурсивною процедурою.

До цих процедур передається параметр-значення, що є покажчиком на корінь дерева. Тіло всіх трьох процедур містить однаковий набір операторів. Першою виконується перевірка того, чи не є дерево порожнім. Якщо дерево порожнє, здійснюється рекурсивне повернення, а в іншому разі виводиться значення вузла і рекурсивно викликаються процедури обходу для лівого та правого піддерев. Порядок цих трьох операторів і визначає форму виразу, що буде створений у результаті обходу. А саме, якщо виведення значення вузла виконуватиметься першим, то буде отримано префіксну форму виразу, якщо другим — інфіксну, а якщо третім — постфіксну форму.

```
{============= обхід дерева зверху вниз ==============}
procedure Prefixorder(RootTree:ptr);
begin
  if RootTree<>nil then
  begin
    write(RootTree^.data,' ');
    Prefixorder(RootTree^.left);
    Prefixorder(RootTree^.right);
  end;
end;
{============ обхід дерева знизу вверх ===============}
procedure Postfixorder(RootTree:ptr);
begin
  if RootTree<>nil then
  begin
    Postfixorder(RootTree^.left);
    Postfixorder(RootTree^.right);
    write(RootTree^.data,' ');
  end;
end;
{============= обхід дерева зліва направо ============}
procedure Infixorder(RootTree:ptr);
begin
  if RootTree<>nil then
  begin
    Infixorder(RootTree^.left);
    write(RootTree^.data,' ');
    Infixorder(RootTree^.right);
  end;
end;
```

```
{основна програма обходу дерева}
begin
   writeln('Create and display tree');
   writeln('Enter number of trees nodes');
   readln(n);                    {ввести кількість вузлів}
   root:=Tree(n);                {створити дерево         }
   writeln('Created tree');
   Printtree(root,0);            {вивести дерево          }
   writeln;
   writeln('prefixorder traversal');
   Prefixorder(root);            {обхід зверху вниз       }
   writeln;
   writeln('postfixorder traversal');
   Postfixorder(root);           {обхід знизу вверх       }
   writeln;
   writeln('infixorder traversal');
   Infixorder(root);             {обхід зліва направо     }
   writeln;
   readln;
end.
```



Домашнє завдання

Прочитати с.23-83 [1] , с.310-341 [4]

Підготуватися до виконання практичної роботи №3.

Приклад виконання практичної роботи №3.

Тема: Абстрактні типи даних

Склад звіту:

- постановка задачі (вказати, який АТД досліджується, та які реалізації вибрано);
- блок-схеми реалізацій, на яких виконано аналіз складності алгоритмів (розглянути тільки операції додавання та видалення елемента);
- опис тестових даних (якого характеру дані і для якої перевірки використані);
- результати дослідження у вигляді графіків або діаграм;
- висновки про доцільність використання кожної з реалізацій для типових вхідних даних та про відповідність результатів експериментального дослідження аналітичним оцінкам складності.