Структури даних

Проста змінна, масив, стек, черга

Структура даних - це спосіб представлення інформації.

В алгоритмізації знання різноманітних структур даних, ефективне їх використання мають неабияке значення. Правильне, коректне визначення структур даних, що використовуються в алгоритмі, - це значна частина успіху.

Деякі структури даних визначаються безпосередньо в мовах програмування, які ми використовуємо для реалізації алгоритмів. Наприклад, ви вже знайомі з простими змінними, масивами. А деякі організовуються самими розробниками алгоритмів. Саме з ними ми маємо ознайомитися в цьому розділі.

Можна сказати, що структури даних є одним з інструментів розробки алгоритмів. Багато відомих методів алгоритмів базуються на тих чи інших структурах.

Ознайомлюючись із структурами даних, ми, в першу чергу, будемо виділяти два такі важливі моменти:

- відображення структури даних на пам’ять комп’ютера;
- обробка інформації в даній структурі.

Перше питання пов’язане з визначенням принципу збереження даних у пам’яті комп’ютера, а друге - з організацією читання та запису інформації. Слід звернути увагу на те, що в різній літературі читання інформації трактується як її вилучення.

Також не менш важливою є інформація про доступ до елементів кожної структури: прямий чи послідовний.

Якщо до будь-якого елемента заданої структури можна дістатися, не переглядаючи всі, то такий доступ називають ***прямим***. У протилежному випадку - ***послідовним***.

ПРОСТА ЗМІННА

Під час виконання програми, в якій визначено просту змінну, операційною системою в пам’яті комп’ютера відводиться місце для її значення. При цьому визначається адреса, за якою вона буде розташована, та частина підряд розташованих комірок, кількість яких відповідає визначеному для цієї змінної типу.

Отже, тепер протягом виконання програми з ім’ям описаної змінної пов’язана конкретна адреса в пам’яті комп’ютера та відповідна до її типу кількість байтів, що починаються з цієї адреси.

Ми визначилися з відображенням простої змінної на пам'ять комп’ютера. Тепер розглянемо, як відбувається запис і читання значень цієї найпростішої структури даних.

При виконанні дій, робота яких пов’язана із записом значення змінної у відповідну область пам’яті, тобто стандартних процедур **read, readln** та оператора присвоювання змінній деякого значення, відбувається визначення адреси, що відповідає розміщенню значення даної змінної в пам’яті комп’ютера і напис указаного двійкового значення за цією адресою.

Виконання операції читання значення простої змінної відбувається при використанні стандартних процедур **write, writeln** та обчислення значень виразів, у яких використовуються ідентифікатори цих змінних. При цьому система «знає» адресу розміщення в пам’яті значення цієї змінної і «читає» його за цією адресою. Далі відбувається процес декодування двійкового зображення інформації, що відповідає правилу її кодування згідно з указаним для цієї змінної типом.

Можна зробити висновок, що ***прості змінні є структурою прямого доступу***.

МАСИВ

З масивами ми вже ознайомилися раніше і набули певного досвіду роботи з ними. Тому нашим завданням є переважно уточнення питання розташування масивів у пам'яті комп'ютера та алгоритму доступу до їх елементів при виконанні операцій читання і запису.

Розглянемо спочатку *одновимірні* масиви.

Спершу нагадаємо, що пам'ять комп'ютера є дискретною і має лінійну структуру. Тобто, всі комірки пам'яті мають абсолютні адреси, що починаються з 0. Для зручності адреси комірок пам'яті комп'ютера представляють шістнадцятковими числами: 0, 1, 2, ..., 9, A, B, C, D, E, F, 10, ..., FF, ...

Саме тому відображення елементів одновимірного масиву на пам'ять комп'ютера уявити собі зовсім нескладно: вони записані в пам'ять підряд, починаючи з деякого визначеного місця.

При розташуванні масиву a1, a2, ..., an у пам'яті комп'ютера визначається адреса його початку, тобто першого елемента масиву a1, і далі розміщуються всі елементи, займаючи кожний стільки байтів, скільки визначено вказаним типом. Тобто, система «знає» адресу першого елемента масиву, скільки байтів займає кожний елемент і загальну кількість елементів масиву.

Адреса розміщення будь-якого іншого елемента, аі визначається за таким алгоритмом: до адреси першого елемента додається кількість байтів, що займає кожний елемент, помножена на порядковий номер потрібного елемента. Таким чином, існує алгоритм обчислення адреси будь-якого елемента одновимірного масиву. Перевага такого принципу безперечна: системі не треба знати адресу кожного елемента масиву, оскільки її можна обчислити за допомогою елементарних арифметичних дій.

Тепер зрозуміло, як відбувається читання і запис інформації, коли використовується структура даних «масив». Спершу визначається адреса потрібного елемента масиву, а потім дії читання і запису виконуються так само, як і для простих змінних.

Можемо також зробити висновок, що ***структура даних «масив»*** є, як і проста змінна, ***структурою прямого доступу.***

Розташування елементів *двовимірного* масиву aij, де i = 1, 2, ..., n; j = 1, 2,,..., m у пам’яті комп’ютера відбувається так:

спочатку розміщуються елементи першого рядка, потім другого і т. д., при визначенні місця розташування елементів двовимірного масиву стає відомою адреса першого елемента. Адреса будь-якого іншого елемента aij цього масиву визначається за алгоритмом:

<адреса aij> = <адреса a11> + ((i - 1) •m + j - 1) •<кількість байтів указаного типу>.

Тобто для визначення адреси поточного елемента двовимірного масиву потрібно до адреси першого елемента додати кількість байтів, яку займає один елемент указаного типу, помножену на кількість елементів двовимірного масиву (i - 1) • m + j — 1, що розміщені в пам’яті комп’ютера від першого елемента до даного aij.

Елементи масивів більшої вимірності розміщуються в пам'яті комп’ютера аналогічно.

Наприклад, *тривимірний* масив зручно уявити собі як паралелепіпед, кожен шар якого представляє собою таблицю, тобто двовимірний масив. Отже, тривимірний масив є масивом двовимірних масивів. У пам’яті комп’ютера він виглядатиме так: елементи першого «шару» тривимірного масиву будуть розташовані в пам’яті за правилом двовимірного масиву, потім так само елементи другого «шару» і т. д.

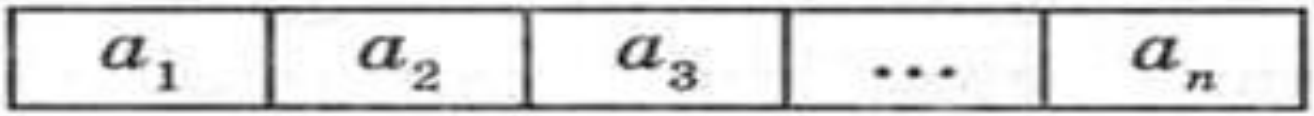
СТЕК

Стеком називається структура даних, що організована за принципом «останнім прийшов — першим пішов».

З поняттям стека ми вже зустрічалися при ознайомленні з алгоритмом роботи рекурсії.

Ми домовилися розглядати структури даних у відображенні на пам’ять комп’ютера. Тому в першу чергу треба визначитися, яким чином елементи даної структури зберігаються в пам’яті.

Оскільки організація одновимірного масиву аналогічна лінійній структурі пам’яті комп’ютера, то логічним буде представлення стека саме у вигляді *одновимірного масиву* (мал. 5).



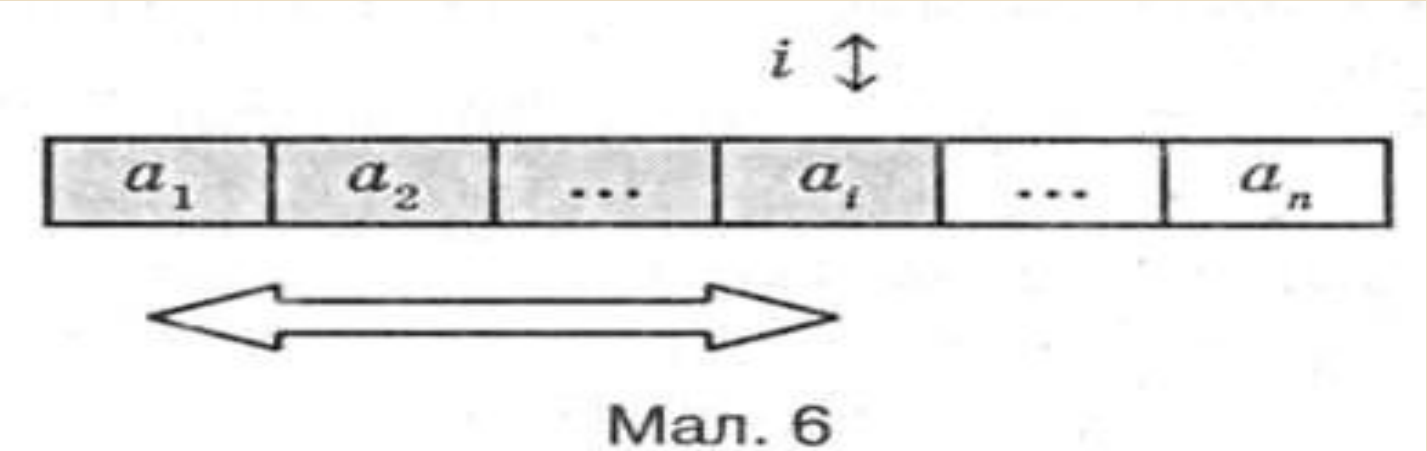
Мал. 5

А от організація обробки елементів цього масиву буде такою, що відповідає означенню цієї структури даних. Для простішого розуміння поняття «стек» проведемо аналогію зі стосом книг.

Додаючи нову книжку в стос, ми кладемо її зверху, тобто операція запису нового елемента в стек відбувається в кінець масиву, додаванням нового елемента. А для того щоб узяти книжку із середини стосу, нам треба всі верхні книжки по одній зняти у зворотному порядку щодо їх складання. Таким чином, операція читання зі стека відбувається також із кінця масиву, але в зворотному порядку.

Обробляючи елементи масиву, ми вводимо поняття поточного порядкового номера елемента масиву i. В даному випадку нас цікавитиме лише останній елемент: ми його або зчитуємо (↑), або після нього записуємо новий елемент (↓).

Індекс останнього елемента стека називається ***вершиною***. Це означає, що для обробки елементів стека нам достатньо знати значення лише однієї величини і вершини стека. Схематично стек можна зобразити так (мал. 6):



Мал. 6

Подвійною стрілкою (↕) ми позначили елемент вершини стека, який доступний як для читання, так і для запису, а сірим кольором - ті елементи масиву, які належать стеку. Нижня стрілка (←→) показує зміну розміру стека в межах визначеного масиву.

Алгоритм ***запису*** нового елемента *x* у стек виглядатиме так:

i:=i+1;

ai:= x.

Тобто ми спочатку збільшуємо значення вершини стека на 1, а потім елементу масиву з індексом і присвоюємо значення x.

Алгоритм ***читання*** чергового елемента стека буде таким:

x:= ai;

i:=i-1.

Пояснимо цей алгоритм: спочатку значення елемента стека, що знаходиться на його вершині *i*, читаємо в *x*, а потім зменшуємо значення вершини стека на 1.

Тепер, коли ми визначили операції для роботи з елементами *i* тільки обов'язково треба зауважити таке.

Для швидкодії роботи зі стеком не витрачається час на «знищення» елементів стека, які вже «вичитано». Доступ до цих елементів утрачається лише за рахунок зміни значення вершини стека (*i* - 1). Таким чином, у масиві лишається «сміття». При наступному «попаданні» на ці елементи в масиві під час виконання операції запису туди будуть записані нові значення.

Розглянемо тепер реалізацію цих алгоритмів мовою Pascal. Перш за все треба обговорити критичні ситуації, які можуть виникнути при читанні та записі інформації. А саме: при читанні може виникнути ситуація, коли вже читати немає чого, тобто стек порожній, а при записі — стек переповнений, тобто досягнуто останнього елемента масиву, оскільки, описуючи масив у програмі, ми повинні вказати межі зміни його індексу. Процедура читання зі стека:

```
procedure read_from_stack;
begin
if i = 0 then writeln('Стек пустий')
else
   begin
writeln(a[i]);
dec(i)
   end
end;
```

Логічним є твердження, що на початку роботи програми з обробки елементів стека вершина стека повинна мати значення 0, оскільки в початковому стані дозволена лише операція запису в стек.

Процедура запису в стек може виглядати так:

```
procedure writetostack;
begin
if i > = n then writeln('Стек повний')
else
    begin
inc(i)
readln(a[i]);
    end
end;
```

При виконанні програми, що реалізує роботу зі стеком, нам цікаво спостерігати за вмістом стека під час виконання операцій читання та запису. Для цього можна скористатися процедурою виведення поточного вмісту стека:

```
procedure print_stack;
var k: word;
begin
for k := 1 to i do
write(a[k], ”);
      writeln
end;
```

Корисною також є інформація про вміст усього масиву, відведеного для організації стека, під час обробки елементів стека. Саме при виконанні цієї процедури можна спостерігати за появою «сміття» в масиві та заміною його на нові значення при виконанні операції запису в стек.

```
procedure printmas;
var i: word;
begin
for i := 1 to n do
write(a[i], ’ ’);
      writeln
            end;
```

Нам залишилося ще визначитися з принципом доступу до елементів даної структури. Робота зі стеком зводиться до обробки вершинного елемента, який доступний завжди, а решта елементів - ні. Щоб зробити доступним деякий елемент a_k ($k < i$), треба спочатку вичитати зі стека всі елементи, що знаходяться вище від нього, зробивши таким чином даний елемент вершиною стека.

Тому можна зробити висновок: ***структура даних «стек» є структурою послідовного доступу.***

Отже:

Стек - це впорядкована лінійна динамічно змінювана послідовність елементів, в якій виконуються наступні умови:

1) новий елемент приєднується завжди до одного і того ж боку послідовності;

2) доступ до елемента здійснюється завжди з того боку послідовності, до якого приєднується елемент;

3) елемент зберігається в послідовності до моменту його виклику .

Операцію включення елемента у стек називають "проштовхуванням", а операцію виключення із послідовності - "виштовхуванням". Найбільш і найменш доступні елементи стека навивають відповідно верхом і низом стека. Операції включення і виключення зі стека виконують в одному і тому ж боку послідовності, але виключаються елементи зі стека в порядку, зворотному до того, в якому вони потрапили в послідовність. В кожний момент часу зі стека можна забрати лише один елемент. Дисципліну обслуговування стека часто називають дисципліною ***LIFO*** - "останній прийшов, перший пішов".

Стек можна зобразити у вигляді послідовності $A=A_1,...,A_{n-1},A_n$, де A_n - елемент, що вказує на вхід-вихід у послідовність; A_1 , A_n - відповідно низ і верх стека. Щоб прочитати елемент A_j , необхідно вибрати зі стека елементи $A_n,..., A_{j+1}$.
Також стек можна представити у вигляді трубки з підпруженим дном, що розміщена вертикально. Верхній кінець трубки відкритий, у нього можна включати, або заштовхувати елементи. Новий елемент, що додається, проштовхує елементи, включені в стек раніше, на одну позицію вниз. При виключенні елементів зі стека вони виштовхуються нагору.
Незалежно від того, як реалізований стек, процедури, що працюють із ним, повинні вміти поміщати елементи в стек, вибирати елементи зі стеку й перевіряти, чи не порожній стек. Для того щоб можна було працювати з новим типом даних, необхідні спеціальні операції, наприклад:

NEW (стек) створення порожнього стека
PUSH (стек, елемент даних) розміщення елемента в стек
POP (стек) вибірка елемента з вершини стека
EMPTY (стек) перевірка, чи порожній стек;
результатом є значення "істина", якщо стек порожній, і "хибно " в протилежному випадку.

NEW і PUSH - базові операції, використовувані при роботі зі стеком. Перша створює стек, а друга поміщає в нього елементи.
POP - процедура, характерна тільки для роботи зі стеком. Вона може бути визначена за допомогою двох базових проміжних операцій TOP і REMOVE, які дозволяють виділити дві основні функції, реалізовані POP, але не визначають способів організації доступу до структури з інших модулів:

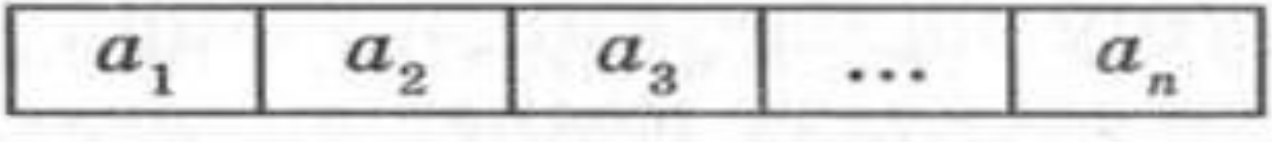
Відповідно наведеним вище визначенням, POP вибирає вершину стека й видаляє її зі стека. Вершина стека описується в такий спосіб: якщо стек не порожній, вершина являє собою останній розміщений у стек елемент; а якщо ні, то стек не має вершини. Операція REMOVE дає в результаті стек без елемента, розміщеного в нього останнім. До тільки що створеного стеку операцію REMOVE застосовувати не можна. Для визначення порожнього стека використовується поняття нового (або тільки що створеного) стека: порожній стек не містить жодного елемента.

Будь-яка реалізація стека повинна задовольняти цим визначенням. Важливо, що стек визначається як спосіб зберігання даних, які підпорядковуються певним правилам, що діють при звертанні до даних, а не як масив з вказівниками, які пересуваються за певними правилами.

Основною перевагою стека перед іншими організаціями даних є те, що у ньому не потрібна адресація елементів. Для обслуговування стека потрібно лише дві команди: PUSH - "проштовхнути" і POP - "виштовхнути". Зі стеком пов'язаний завжди один вказівник, який вказує на верхній елемент у стеку. На початку такий вказівник дорівнює нулю.
Найпростішим прикладом вдалого застосування стека може служити задача перепису вектора у зворотному порядну. На практиці стекова структура даних найчастіше застосовується в рекурсивних алгоритмах, при трансляції, а також при обробці переривань програм.

Чергою називається структура даних, організована за і принципом «першим прийшов — першим пішов».

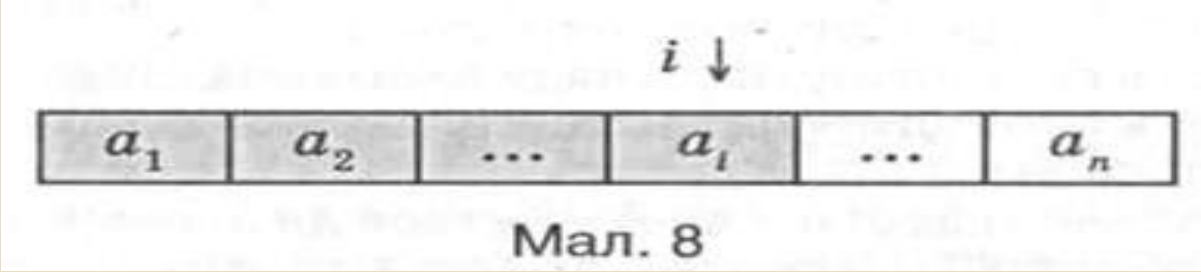
Черга, так само як і стек, відображається на пам’ять комп’ютера у вигляді одновимірного масиву (мал. 7).



Мал. 7

Принцип обробки елементів черги схожий на звичайне формування черги в магазині. В кожний момент часу обслуговується перший покупець. Після обслуговування він іде з черги і на його місце стає наступний - тепер він перший. Новий покупець стає в кінець черги.

Якщо цю схему перенести на обробку елементів черги, то, на перший погляд, треба знати лише індекс останнього елемента черги, щоб після нього можна було записати новий елемент. Початок черги при цьому завжди буде збігатися з першим елементом масиву (мал. 8).

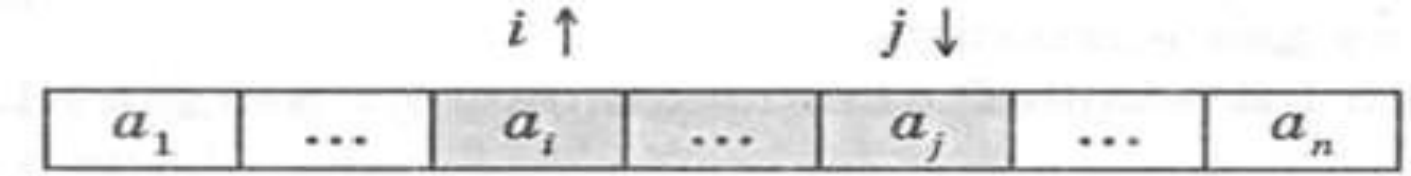


Мал. 8

Але при такій логіці існує один суттєвий недолік: після вибування з черги першого елемента всі решта повинні зсуватися на одну позицію вліво. Тобто потрібно кожний раз при виконанні операції читання виконувати таку послідовність дій:

ai:= ai + 1 для i = 1, 2, ..., <індекс кінця черги >.

Зрозуміло, що така схема обробки операції читання забирає багато часу. Для його економії застосуємо ідею читання елемента стека. Тобто після читання елемента не будемо намагатися знищити його значення, а просто перемістимо індекс початку черги на наступний елемент. Таким чином ми приходимо до ідеї існування двох індексів: i - початок черги, який ще носить назву «голова» черги, та j - кінець черги, що називається «хвостом» черги. Тепер схематично чергу можна зобразити так (мал. 9):



Мал. 9

На схемі сірим кольором позначена та частина масиву, в якій розміщені елементи черги.

Подамо алгоритм запису в чергу елемента x:

j:=j+1;

aj:=x.

Тобто ми спочатку збільшуємо значення індексу «хвоста» черги на 1, а потім елементу масиву з індексом j присвоюємо значення x.

Алгоритм читання елемента черги виглядатиме так:

x:=ai;

i:=i+1.

Тобто значення «голови» черги читаємо в x, а потім збільшуймо індекс «голови» черги на 1.

Тепер можна уявити собі, що наша черга ніби повзає по масиву, переміщуючись від першого елемента до останнього. Причому елементи черги не розриваються і знаходяться в тій послідовності, в якій записувалися. Можна говорити, що запис у Чергу буде неможливий, тобто черга переповнена, коли «хвіст» досягне останнього елемента масиву, а читання стане неможливим, тобто черга порожня, коли «голова» пережене «хвіст».

Запишемо в термінах такого тлумачення процедуру читання елемента черги:

```
procedure read_from_queue;
begin
if i > j then writeln('Queue is empty')
   else
begin
    writeln(a[i]);
    inc(i)
end
end;
```

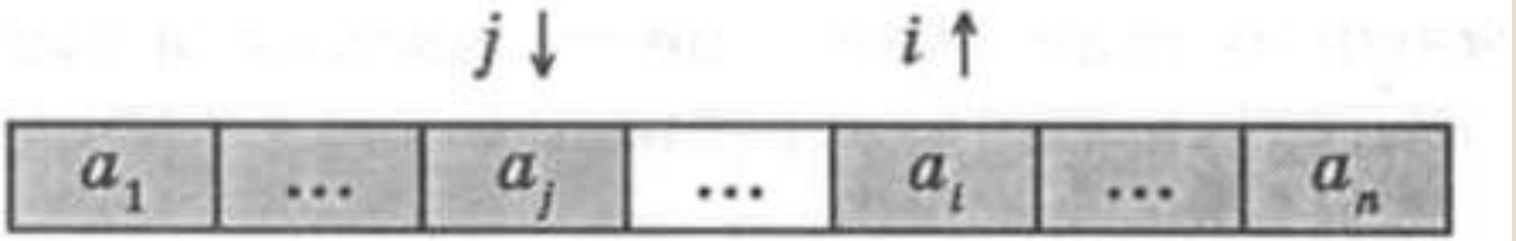
Процедура запису елемента в чергу:

```
procedure write_to_queue;
begin
      if j = n then writeln('Queue is full')
      else
begin
     inc(j);
    readln(a[j])
end
end;
```

Але це досить спрощений погляд на ефективне використання елементів масиву для розміщення черги, адже в масиві можуть залишатися елементи, які не є на даний момент елементами черги. Розглянемо кілька конкретних ситуацій.

Нехай виконуємо лише операції запису в чергу. При цьому «голова» черги весь час знаходиться на першому елементі масиву, а «хвіст» зміщується в сторону останнього. Коли буде досягнуто кінця масиву, це означатиме, що черга переповнена і наступні операції запису стануть неможливими. Це видно з наведених вище процедур запису та читання.

Однак ситуація зміниться, якщо поряд із операціями запису виконуватимемо і операції читання з черги. На перший погляд, запис стане неможливим, коли «хвіст» черги досягне останнього елемента масиву. Але на початку масиву є вільні місця, що залишилися після виконання читання з черги (i > 1). Чому б їх не використати під елементи черги? Для цього треба лише виконати операцію j:= 1. Наша черга ніби завернеться своїм хвостом на початок масиву. Тепер ознакою того що черга переповнена, буде ситуація, коли «хвіст» дожене «голову». Схематично це виглядатиме так (мал. 10):



Мал. 10

Отже, перед тим як розглянути удосконалені процедури запису в чергу та читання з неї, введемо деякі домовленості. Це пов’язане з тим, що конкретна реалізація алгоритму мовою програмування не дає можливості використовувати певні умовності. А саме, ми зазначали вище, що виконання умови i = j свідчить про те, що черга порожня, а j = i- про те, що черга повна. Але з погляду будь-якої мови програмування ці дві умови абсолютно ідентичні, тому для їх «розпізнавання» потрібно ввести додаткові ознаки. Однак нашим завданням не є завершена реалізація всіх алгоритмів, оскільки кожний програміст має свій власний «почерк» реалізації алгоритмів. Тому умову i <= j замінятимемо виразом <«голова» передує «хвосту»>, а j <= i - виразом <«хвіст» передує «голові»>.

Тепер запишемо удосконалену процедуру запису в чергу:

```
procedure write_to_queue;
begin
if (j = i) and (<«хвіст» передує «голові» > )
then writeln('Queue is full’)
else
    begin
readln(a[j]);
if j = n
then
    begin
j:= 1; < «хвіст» передує «голові»>
end
          else inc(j);
     end
end;
```

Аналогічно можна поставити запитання і щодо «голови» черги, коли вона досягне останнього елемента масиву. Якщо і «хвіст» не збігається з початком масиву, тобто там ще є місце для нашої черги, то чому б не почати відлік «голови» з початку масиву. Тобто якщо і > п, то потрібно виконати операцію і := 1. Удосконалимо і процедуру читання з черги:

```
produre read_from_queue;
begin
if (i = j) and <«голова» передує «хвосту»>
then writeln (’Queue is empty’)
else begin
writeln(a[i]); if i = n then
begin
i:=1;<«голова» передує «хвосту»>
end
else inc(i);
end
end;
```

Ми можемо сказати, що удосконалюючи алгоритм обробки елементів черги, зробили її кільцевою.

Процедуру перегляду елементів черги можна записати в такому вигляді:

```
procedure print_queue;
var k: word;
begin
if <«голова» передує «хвосту»>
then for k := i to j - 1 do write(a[k], ' ')
else begin
for k := i to n do write(a[k], ' ');
for k := 1 to j - 1 do write(a[k], ' ');
  end;
writeln
end;
```

Залишилося лише визначитися з принципом доступу до елементів черги. Зрозуміло, що в кожний момент часу ми можемо здійснювати операції читання та запису над елементами черги, що знаходяться в «голові» або в «хвості» черги. Дістатися до деякого «внутрішнього» елемента черги ми можемо, лише виконавши відповідну кількість операцій читання з «голови» черги. Тому можна зробити висновок, що черга є структурою послідовного доступу.

Отже:

Черга - це лінійна динамічно змінювана послідовність елементів, у якій виконуються такі умови:

1) новий елемент приєднується завжди з одного і того ж боку послідовності;

2) доступ до елементів або їх виключення завжди здійснюється з другого боку послідовності.

Наприклад, нехай послідовність $Q=Q_1... ,Q_n$ зображує чергу. Тоді елемент Q_1, називають "головою" черги і він вказує на місце для виключення елемента; а елемент Q_n називають "хвостом" черги і він вказує на місце для включення елемента в чергу.

Чергу можна представити у вигляді трубки. В один кінець трубки можна додавати кульки — елементи черги, з іншого кінця вони видаляються. Елементи в середині черги, тобто кульки усередині трубки, недоступні. Кінець трубки, у який додаються кульки, відповідає кінцю черги, кінець, з якого вони видаляються — початку черги. Таким чином, кінці трубки не симетричні, кульки усередині трубки рухаються тільки в одному напрямку.

Отже, в структурі черг потрібні два вказівники: один - для посилання на вершину послідовності (для включення елемента), другий - для посилання на основу послідовності (для виключення елемента). При кожному виключенні елемента із такої структури виключається завжди найстаріший елемент. Черги обслуговуються за дисципліною ***FIFO*** - "перший прийшов, перший пішов". Черги бувають різних типів: лінійні, з пріоритетом і циклічні. Звичайна **лінійна** черга може бути зображена масивом, з двома вказівниками: перший вказує на елемент для вибірки з черги, другий - на останній елемент, записаний у чергу. Багаторазове звертання до елементів лінійної черги призводить до того, що пам'ять для зберігання такої черги використовується неефективно. У лінійній черзі після вибірки елемента пам'ять, зайнята чергою, звільняється і повторно не використовується, оскільки нові елементи приписуються з іншого краю послідовності.

Чергу, для якої є можливість включати або виключати елементи з певної позиції в залежності від деяких її характеристик називають **пріоритетною** чергою. Прикладом пріоритетної черги може служити порядок розв'язування потоку задач на комп‘ютері у деяких операційних системах . Така черга зводиться до послідовності лінійних черг, якщо відомі пріоритети її елементів. Кожна черга з послідовності обслуговується за дисципліною *FIFO* , але елементи з другої черги обслуговуються тільки тоді, коли порожня попередня черга, а з третьої тоді, коли порожні перша і друга черги. При включенні елементи приєднуються до боку однієї з черг згідно з їх пріоритетом.

Черги, в яких елементи Q_1 ,Q_2, ...,Q_n розміщуються так, що за елементом Q_n іде елемент Q_1 називаються **циклічними** (рис. 7.1). Циклічну чергу також можна зобразити лінійною послідовністю, але при записі нового елемента на відміну від лінійної черги вся послідовність зсувається на одне поле і новий елемент записується знову на її початку. Вибірка елемента буде також виконуватися за вказівником на кінець послідовності.

Прикладом циклічної черги може служити робота обчислювальної системи з розподілом часу, з якою одночасно працює багато користувачів. Оскільки така система має переважно один блок обробки, що називають процесором, і одну пам'ять, то в кожний момент часу ці ресурси належать одному користувачу. Для кожного користувача виділяється певний інтервал часу. Тільки на відміну від пріоритетної черги одна задача до кінця не розв'язується, а "порціями" протягом виділеного їй інтервалу часу. Програми, що чекають на виконання, утворюють циклічну чергу.