Динамічні структури даних. Керування пам'яттю

Де початок того кінця,
яким закінчується початок?
Козьма Прутков

Динамічні структури даних

Довільна програма створюється для обробки даних, від способу організації яких суттєво залежать алгоритми її роботи, тому вибір структур даних має першочергове значення.

Розглядали “стандартні” способи організації даних, що основані на використанні скалярних та структурних типів. Спільне – структури мали фіксований розмір на протязі роботи з ними (масиви, структури, масиви структур, …). Це визначає суттєві обмеження.

Але пам`ять під данні може виділятися або на етапі компіляції (при цьому її розмір фіксується), або у ході виконання програми – областями вказаного розміру (розмір може змінюватись за потребою).

Динамічні структури даних

Виділення пам`яті під час виконання програми надає гнучкість у представлення даних. Пам`ять виділяється, та звільняється блоками (неперервними), що зв`язані за допомогою покажчиків. Такий спосіб організації даних називають ***динамічними структурами даних***, оскільки їх розмір може змінюватись у ході роботи.

Використовують як *лінійні*, так й *нелінійні динамічні структури*:

- лінійні списки (стеки, черги);
- дерева (бінарні, двійкового пошуку, збалансовані, …).

Вказані структури розрізняються як способами зв`язку між окремими елементами структури, так й доступними операціями.

Динамічні структури даних

Динамічні структури дозволяють гнучко й ефективно працювати з даними розмір яких заздалегідь невідомий, а також пришвидшити роботу з даними при виконанні операцій додавання, вилучення, пошуку, впорядкування даних.

Існуючі можливості для ефективного представлення та обробки різноманітної інформації є предметом подальшого розгляду.

Для динамічних структур основні:

- базові типи даних – структури, покажчики, масиви;
- операції – виділення та звільнення пам`яті, доступу до даних через покажчики.

Інструментарій C++

Елемент динамічної структури представляється структурою, що містить принаймні два поля: для збереження даних (довільні типи та кількість полів), для збереження зв'язків (поля покажчиків). Наприклад:

```
struct Node {
        Data d; //тип Data повинен бути визначений раніше
    Node *p; }
Node *newPtr;
```

Основні операції в C++ new , delete :

```
newPtr = new Node;
delete newPtr;
```

Звернення до даних:

```
newPtr -> d     або    (*newPtr).d
```

Інструментарій C++

Для роботи з адресами пам`яті C++ пропонує досить чисельний набір інструментів.

Розглядали раніше операції (***повторити***):

- унарні: ***** *-(розіменування)*, **&** *-(отримання адреси)*, **new** , **delete** -(виділення та звільнення пам`яті);
- бінарні: **.**, **->** (*отримання поля, розіменування поля*).

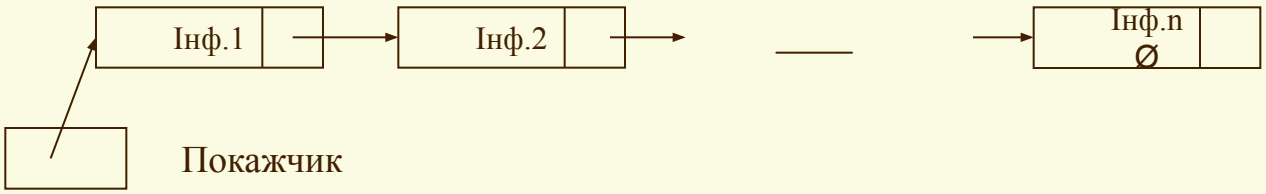
Стандартна бібліотека (<stdlib.h>) надає функції для керування розподілом пам`яті (***самостійно***):

- **calloc()**, **malloc()**, **realloc()** – виділення;
- **free()** – звільнення.

В C++ рекомендовано для керування розподілом пам`яті використовувати операції **new** та **delete** .

Лінійні зв`язані списки

Найпростіші динамічні структури даних – лінійні зв`язані списки. Як мінімум зберігається зв`язок поточного вузла з наступним. Список задається покажчиком на початковий вузол.

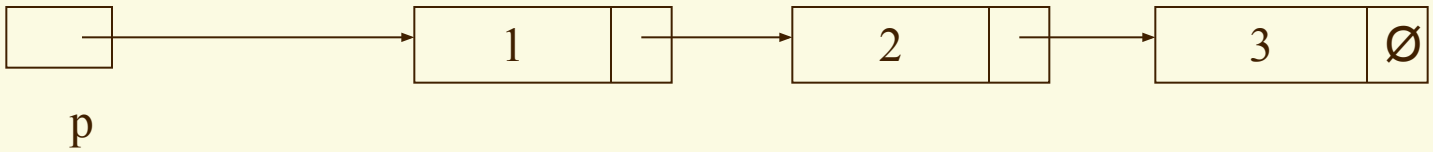


!!! Втрата покажчиків (розрив зв`язків) призведе до недоступності відповідних елементів, або списку в цілому (поява “сміття”).

Приклад

Потрібно:

- побудувати зв`язний список з трьох елементів <1, 2, 3>;
- вивести інформацію, що розташована в елементах списку;
- звільнити пам`ять, прибравши елементи списку.



В прикладі розглянемо покрокове виконання всіх вказаних дій зі списком.

Представлення даних

```
typedef struct Node {int dat;
                     Node *next;} Listn, *Listp;
```

!!! Не єдиний можливий спосіб визначити потрібні типи даних.

Побудова списку

```
//побудова списку з 3 елементів
Listp lform(int d1, int d2, int d3){
  Listp p, t;
  p = new Listn;  p->dat = d3; p->next = NULL;
  t = new Listn;  t->dat = d2; t->next = p; p = t;
  t = new Listn;  t->dat = d1; t->next = p; p = t;
  return p;
}
```

Відображення списку

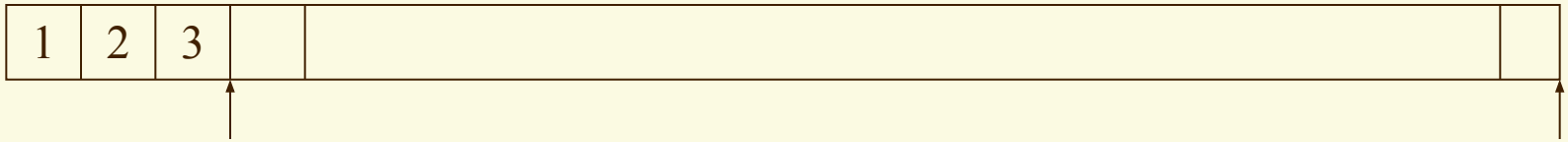
```
//відображення 3-елементного списку
void lprint(Listp p){
   cout << p->dat << ' '; p = p->next;
   cout << p->dat << ' '; p = p->next;
   cout << p->dat << endl;
}
```

Вилучення списку

```
//знищення 3-елементного списку
void ldel(Listp p){
   Listp t;
   t = p->next; delete p; p = t;
   t = p->next; delete p; p = t;
   delete p;
}
```

Альтернативи

Для представлення списку використовувати масив.



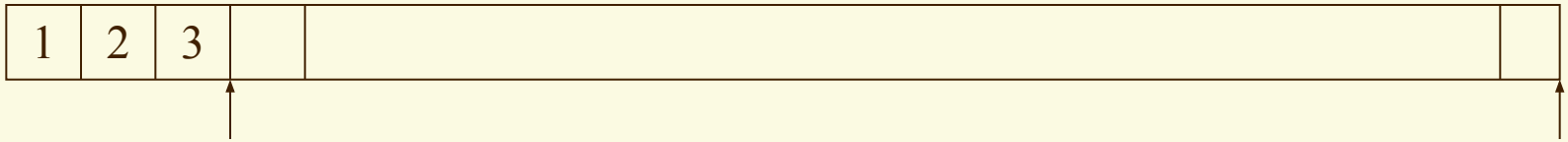
Не є проблемою записати відповідний аналог розглянутим діям.

Порівнянні наведених способів.

???

Альтернативи

Для представлення списку використовувати масив.



Порівнянні наведених способів:

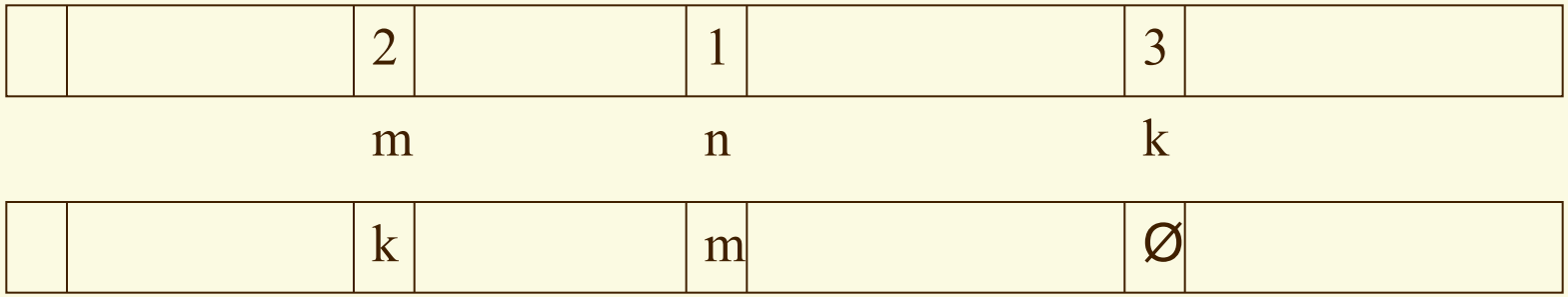
-фіксований розмір масиву → обмеження на розмір списку, ефективність використання наявної пам`яті;

-зв`язки, порядок елементів списку визначається розташуванням у масиві → зміни – передбачають переміщення елементів;

-масив займає неперервну область пам`яті;

-не завжди мови програмування надають явні можливості роботи з адресами.

Альтернативи



beg | n |

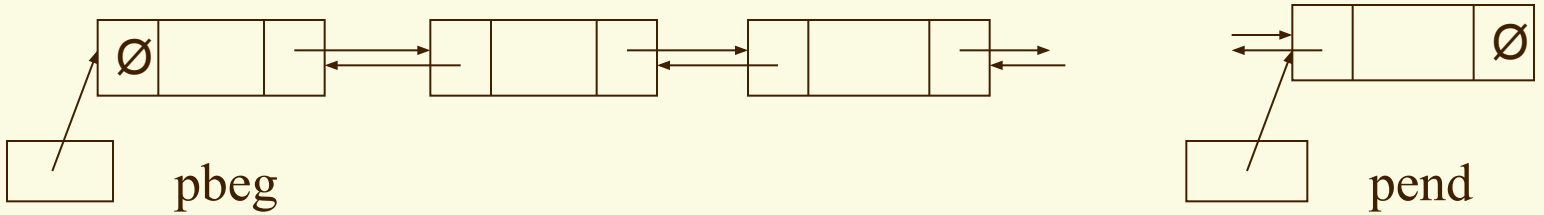
Використовує явно лише можливості роботи з масивами та їх елементами.

Порівнянні наведених способів.

???

Приклад

Використовують також списки з двома зв`язками.



Потрібно:

Їх переваги, вади ???

- побудувати подібний список з N елементів <1, 2, …, N>;
- додати новий елемент з заданим значенням після елемента з вказаним значенням (пошук місця вставки);
- вилучити елемент з заданим значенням (пошук ел.);
- вивести список на екран;
- знищити список, звільнивши пам`ять.

Представлення даних

```
struct Node {int dat;
            Node *next;
            Node *prev;};
```

!!! Не єдиний можливий спосіб визначити потрібні типи даних.

Побудова списку

```
//формування першого елемента списку
Node *first(int d){
    Node *pv = new Node;
    pv->dat = d; pv->next = NULL; pv->prev = NULL;
    return pv;
}
//додавання елементів в кінець списку 2, 3, ..., nn
void add(Node **pend, int d){
    Node *pv = new Node;
    pv->dat = d; pv->next = NULL; pv->prev = *pend;
    (*pend)->next = pv;
    *pend = pv;
}
```

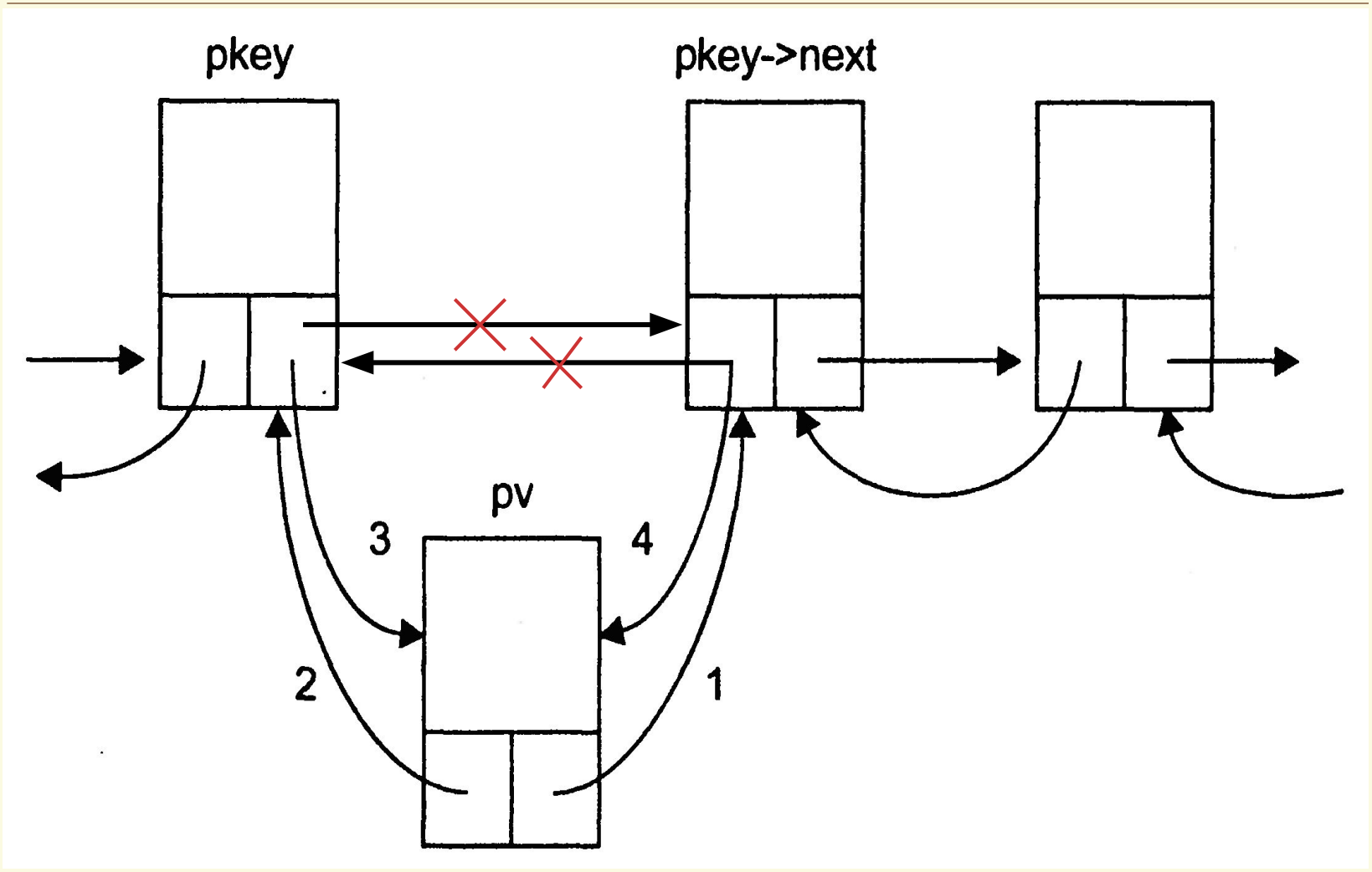
Пошук у списку

```
//пошук елемента за ключем
Node *find(Node *const pbeg, int key){
    Node *pv = pbeg;
    while (pv){
     if (pv->dat == key) break;
     pv = pv->next;
     }
    return pv;
}
```

Вставка елемента у список

```
Node *insert(Node *const pbeg, Node **pend, int key, int d)
{if (Node *pkey = find(pbeg, key))    {
      Node *pv = new Node;
      pv->dat = d;
      pv->next = pkey->next; //зв`язок нового вузла з наступним
      pv->prev = pkey;  //зв`язок нового вузла з попереднім
      pkey->next = pv;  //зв`язок попереднього з новим вузлом
      //зв`язок наступного з новим вузлом
      if (pkey != *pend) (pv->next)->prev = pv;
      else *pend = pv;  //якщо вузол стає останнім
            return pv;}
    return NULL;  //місце для вставки не було знайдено
}
```

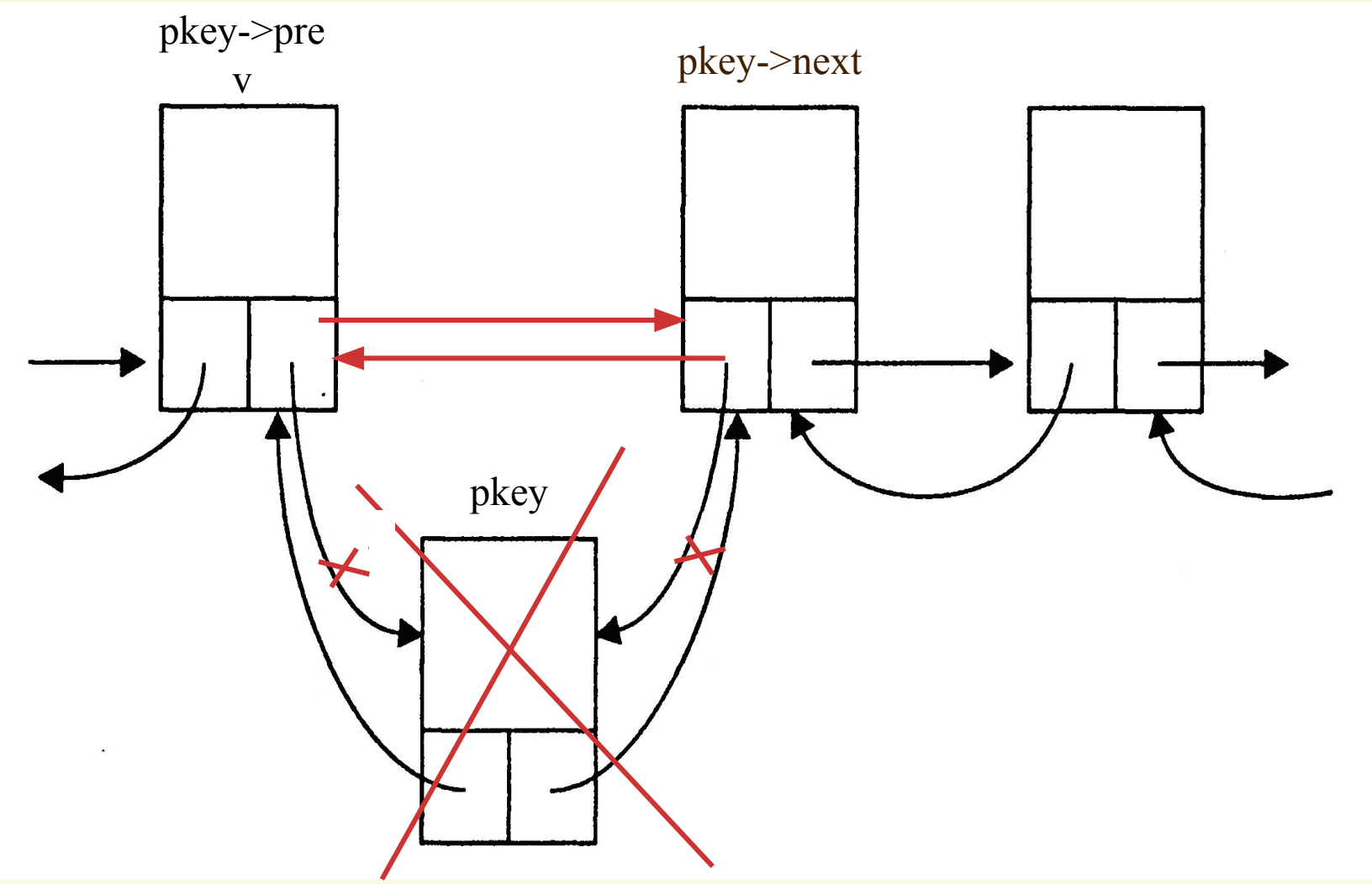
Вставка елемента у список



Вилучення елемента зі списку

```
//вилучення елемента
bool remove(Node **pbeg, Node **pend, int key){
    if (Node *pkey = find(*pbeg, key)){
     if (pkey == *pbeg) { *pbeg = (*pbeg)->next;
                (*pbeg)->prev = NULL;}
            else if (pkey == *pend) { *pend = (*pend)->prev;
                    (*pend)->next = NULL;}
            else { (pkey->prev)->next = pkey->next;
        (pkey->next)->prev = pkey->prev;}
        delete pkey;
        return true;}
    return false;
}
```

Вилучення елемента зі списку



Відображення списку

```
//виведення списку
void lprint(Node *pbeg){
    Node *pv = pbeg;
    while (pv){
     cout << pv->dat << ' ';
     pv = pv->next;
    } cout << endl;
}
```

Вилучення списку

```
//знищення списку
void ldel(Node *pbeg){
    Node *pv;
    while (pbeg){
     pv = pbeg;
     pbeg = pbeg->next;
     delete pv;
    }
}
```

Зауваження

- Стандартна бібліотека C++ надає також інші можливості керування розподілом пам`яті.
- Не відбувається автоматичне повернення виділеної пам`яті, що може стати причиною появи “сміття”, “переповнення” пам`яті.
- Операція delete звільняє пам`ять (яка може бути розподілена у подальшому), але не прибирає й не змінює сам покажчик.
- Динамічні зв`язані структури дозволяють використовувати для представлення великих даних не обов`язково неперервну область пам`яті.

Підсумки

- Розглянули лише найпростіші можливості, що до створення та обробки зв`язаних лінійних списків.
- Розглянуті можливості, приклади будуть виступати базою для подальшого вивчення динамічних структур даних.
- Застосування динамічного розподілу пам`яті, замість масивів, для структур даних, які можуть зменшуватись та збільшуватись у розмірах під час обробки, сприяє більш раціональній організації роботи та заощадженню таких ресурсів, як пам`ять, а іноді й час.

Поради

- Одразу звільняти пам`ять (операція - delete), що була виділена (операцією new) та стала непотрібною.
- Писати “прозорі”, структуровані програми.
- Здійснювати внутрішнє документування у програмі за допомогою коментарів.
- Не зловживати “трюкачеством”.
- Розумним чином “форматувати” текст програми.
- Принаймні на початковому етапі не нехтувати можливостями візуально відслідковувати дії, що відбуваються з динамічною структурою.
- При тестуванні та налагодженні додатково відображати проміжні стани динамічних структур.

Задачі

- Лінійний зв`язний список містить послідовність цілих чисел. Написати функцію для:
  - знаходження максимального з чисел;
  - знаходження кількості чисел у послідовності;
  - перевірки належності заданого числа до послідовності;
  - перевірки наявності двох однакових чисел у послідовності.

Задачі

- Рядки тексту являють собою прізвища, які можуть повторюватися. Потрібно прочитати текст і надрукувати кожне прізвище по одному разу. Порядок прізвищ не має значення. Проблеми:
  - джерело інформації - файл (текстовий);
  - кількість прізвищ не відома.
  - Порада – для тимчасового внутрішнього збереження інформації скористатися динамічною структурою.

Задачі

- У файлі зберігається послідовність цілих чисел. Написати функції потрібні для побудови впорядкованого списку елементів заданої послідовності.
- Написати функцію для зчеплення двох лінійних однозв`язних списків (результат – список, отриманий в результаті додавання в кінець першого списку елементів з другого).