МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Одеська національна академія зв'язку ім. О.С. Попова

Кафедра інформаційних технологій

АЛГОРИТМИ ТА СТРУКТУРИ ДАНИХ

Конспект лекцій для студентів напряму
6.050103 – Програмна інженерія

Частина 1. Структури даних

Одеса 2017

УДК 004.43 План НМВ 2017 р.

Рецензент: к.ф-м.н., доцент кафедри Математичного забезпечення комп'ютерних систем ОНУ ім. І.І. Мечнікова, Антоненко О.С.

Алгоритми та структури даних: конспект лекцій.
Частина 1. Структури даних / Укладачі: О.Д. Воробйова, Л.В. Глазунова – Одеса:ОНАЗ ім.О.С. Попова, 2017. - 48с.

Конспект лекцій містить короткі теоретичні відомості зі структур даних та основних алгоритмів обробки даних, таких як пошук елемента у лінійних структурах чи пошук елемента у тексті, сортування масиву, стиснення даних, пошук найкоротшого шляху. Алгоритми та структури даних – найважливіший розділ комп'ютерних наук і програмування. Завдяки розвитку інформаційних технологій та алгоритмам ми сьогодні маємо можливість швидко знаходити інформацію в Інтернеті (зокрема, шукати по картинках), знаходити найкоротші шляхи, аналізувати геноми і т. д. Алгоритми використовуються практично в усіх областях комп'ютерних наук – в аналізі зображень, Інтернет-пошуку, машинному навчанні, біоінформатиці, криптографії, кодуванні, мережах, розподілених системах, компіляторах.

Конспект лекцій призначено студентам для здобуття теоретичних і практичних знань з напряму 6.050103 – Програмна інженерія.

СХВАЛЕНО
на засіданні кафедри
інформаційних технологій
та рекомендовано до друку.
Протокол № 4
від 28 грудня 2016 р.

ЗАТВЕРДЖЕНО
методичною радою
академії зв'язку.
Протокол № 5
від 28.02.2017 р.

ПЕРЕДМОВА

Дисципліна «Алгоритми та структури даних» присвячена формуванню наукової системи мислення, вмінню проектувати алгоритми і структури даних. Перед вивченням даного навчального курсу передбачається, що студенти володіють знаннями в області найпростіших алгоритмів, володіють мовою програмування високого рівня, володіють основами об'єктно-орієнтованого підходу в аналізі і проектуванні систем.

Поняття "алгоритм і структура даних" є центральним у сфері комп'ютерних технологій, однак, щоб називати деякі структури даних і алгоритми якісними та ефективними, слід використовувати точні прийоми їх аналізу. Як природний критерій якості можно виділити, по-перше, час виконання. Також важливим є обсяг витрачених ресурсів пам'яті і дискового простору. Увагу також слід приділити надійності і достовірності рішень, їх стабільності.

У результаті вивчення першої частини навчальної дисципліни студент повинен *знати:*

- динамічні структури даних;
- динамічні структури - односпрямовані списки;
- динамічні структури - двоспрямовані списки;
- динамічну структуру - стек ;
- динамічну структуру - черга;
- динамічні структури - циклічні (кільцеві) списки;
- динамічні структури - бінарні дерева;

вміти:

- створювати динамічну структуру даних;
- виділяти пам’ять під екземпляри динамічної структури та вивільняти цю пам’ять;
- програмувати основні операції для обробки динамічних структур даних: друк (перегляд), включення елемента, виключення елемента, пошук елемента, перевірка відсутності елементів (NULL).

Лекція 1
ДИНАМІЧНІ СТРУКТУРИ ДАНИХ

У мові С++ є засоби створення динамічних структур даних, які дозволяють під час виконання програми створювати об'єкти, виділяти для них пам'ять, звільняти пам'ять, коли в ній зникає необхідність. Якщо до початку роботи з даними неможливо визначити, скільки пам'яті потрібно для їх зберігання, пам'ять слід розподіляти під час виконання програми в міру необхідності окремими блоками. Блоки зв'язуються один з одним за допомогою покажчиків. Такий спосіб організації даних називається динамічною структурою даних, оскільки вона розміщується в динамічній пам'яті і її розмір змінюється під час виконання програми.

Динамічні структури даних − це структури даних, пам'ять під які виділяється і звільняється в міру необхідності.

Динамічні структури даних в процесі існування в пам'яті можуть змінювати не тільки число складових їх елементів, а й характер зв'язків між елементами. При цьому не враховується зміна вмісту самих елементів даних. Така особливість динамічних структур, як мінливість їх розміру і характеру відносин між елементами призводить до того, що на етапі створення машинного коду програма-компілятор не може виділити для всієї структури в цілому ділянку пам'яті фіксованого розміру, а також не може порівнять з окремими компонентами структури конкретні адреси. Для вирішення проблеми адресації динамічних структур даних використовується метод, називаний динамічним розподілом пам'яті, тобто пам'ять під окремі елементи виділяється в момент, коли вони "починають існувати" в процесі виконання програми, а не під час компіляції. Компілятор в цьому випадку виділяє фіксований обсяг пам'яті для зберігання адреси елемента, який динамічно розміщується, а не самого елемента.

Динамічна структура даних характеризується тим, що:

- вона не має імені;
- їй виділяється пам'ять в процесі виконання програми;
- кількість елементів структури може не фіксуватися;
- розмірність структури може змінюватися в процесі виконання програми;
- в процесі виконання програми може змінюватися характер взаємозв'язку між елементами структури.

Кожній динамічній структурі даних відповідає статична змінна типу покажчик (її значення – адреса цього об'єкта), за допомогою якої здійснюється доступ до динамічної структури.

Самі динамічні величини не вимагають опису в програмі, оскільки під час компіляції пам'ять під них не виділяється. Під час компіляції пам'ять виділяється тільки під статичні змінні. Покажчики – це статичні змінні, тому вони вимагають опису.

Оскільки елементи динамічної структури розташовуються за непередбачуваними адресами пам'яті, адреса елемента такої структури не може

бути обчислена з адреси початкового або попереднього елемента. Для встановлення зв'язку між елементами динамічної структури використовуються покажчики, через які встановлюються явні зв'язки між елементами. Таке уявлення даних в пам'яті називається пов'язаним.

Порядок роботи з динамічними структурами даних наступний:

1) створити (відвести місце в динамічній пам'яті);
2) працювати за допомогою покажчика;
3) видалити (звільнити зайняте структурою місце).

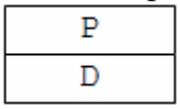
Класифікація динамічних структур даних

У багатьох задачах потрібно використовувати дані, в яких конфігурація, розміри і склад можуть змінюватися в процесі виконання програми. Для їх уявлення використовують динамічні інформаційні структури. До таких структур відносять:

- односпрямовані (однозв'язні) списки;
- двоспрямовані (двозв’язні) списки;
- циклічні списки;
- стек;
- чергу;
- бінарні дерева.

Оголошення динамічних структур даних

Кожна компонента будь-якої динамічної структури є запис, що містить, принаймні, два поля: одне поле типу покажчик, а друге − для розміщення даних. У загальному випадку запис може містити не один, а кілька покажчиків і кілька полів даних. Поле даних може бути змінною, масивом або структурою. Для найкращого уявлення зобразимо окрему компоненту у вигляді:



де поле P – покажчик; поле D – дані.

Елемент динамічної структури складається з двох полів:

– інформаційного поля (поля даних), в якому містяться ті дані, заради яких і створ

– ються структура; в загальному випадку інформаційне поле саме є інтегрованою структурою – вектором, масивом, іншою динамічною структурою і т.п.;

– адресного поля (поля зв'язок), в якому містяться один або кілька покажчиків, що зв'язує даний елемент з іншими елементами структури.

Оголошення елемента динамічної структури даних виглядає наступним чином:

```
struct ім’я_типу {
                    інформаційне поле;
                    адресне поле;
                  };
```

Наприклад:

```
struct TNode {
```

```
    int Data;//інформаційне поле
    TNode *Next;//адресне поле
};
```

Інформаційних і адресних полів може бути як одне, так і декілька. Розглянемо як приклад динамічну структуру, схематично показану на рис. 1.1:

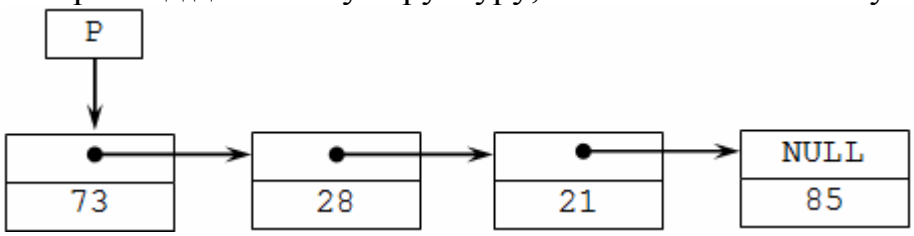


Рисунок 1.1 − Схематичне подання динамічної структури

Дана структура складається з чотирьох елементів. Її перший елемент має поле Data, яке дорівнює 73, і пов'язаний за допомогою свого поля Next з іншим елементом, поле Data якого дорівнює 28, і так далі до останнього, четвертого елемента, поле Data якого дорівнює 85, а поле Next дорівнює NULL, тобто має нульову адресу, що є ознакою завершення структури. Тут P є покажчиком, який вказує на перший елемент структури.

Доступ до даних в динамічних структурах

Елемент динамічної структури в кожен момент може або існувати, або бути відсутнім в пам'яті, тому його називають динамічним. Оскільки елементами динамічної структури є динамічні змінні, то єдиним засобом доступу до динамічних структур та їх елементів є покажчик (адреса) на місце їх поточного розташування в пам'яті. Таким чином, доступ до динамічних даних виконується спеціальним чином за допомогою покажчиків.

Покажчик містить адресу певного об'єкта в динамічній пам'яті. Адреса формується з двох слів: адреса сегмента і зміщення. Сам покажчик є статичним об'єктом і розташований у сегменті даних (рис. 1.2).

Рисунок 1.2 − Зв’язок покажчика з адресованим об’єктом

Для звернення до динамічної структури достатньо зберігати в пам'яті адресу першого елемента структури. Оскільки кожен елемент динамічної структури зберігає адресу наступного за ним елемента, можна, рухаючись від початкового елемента за адресами, отримати доступ до будь-якого елемента даної структури.

Доступ до даних в динамічних структурах здійснюється за допомогою операції "стрілка" (->), яку називають операцією непрямого вибору елемента структурного об'єкта, що адресується покажчиком. Вона забезпечує доступ до

елемента структури через адресу її покажчика того ж структурного типу. Формат застосування даної операції наступний:

ПокажчикНаСтруктуру-> Ім'яЕлемента;

Операція "стрілка" (->) двомісна. Застосовується для доступу до елемента, що задається правим операндом тієї структури, яку адресує лівий операнд. В якості лівого операнда повинен бути покажчик на структуру, а в якості правого − ім'я елемента цієї структури.

Наприклад:

```
p->Data;
p->Next;
```

Робота з пам'яттю при використанні динамічних структур

У програмах, в яких необхідно використовувати динамічні структури даних, робота з пам'яттю відбувається стандартним чином. Виділення динамічної пам'яті проводиться за допомогою операції new або за допомогою бібліотечної функції `malloc (calloc)`. Звільнення динамічної пам'яті здійснюється операцією `delete` або функцією `free`.

Наприклад, оголосимо динамічну структуру даних з ім'ям Node та з полями Name, Value і Next, виділимо пам'ять під покажчик на структуру, дамо значення елементів структури і звільнимо пам'ять:

```
struct Node {char *Name;
             int Value;
             Node *Next
            };
Node *PNode; //оголошується покажчик
PNode = new Node; //видалення пам'яті
PNode->Name = "STO"; //привласнюються значення
PNode->Value = 28;
PNode->Next = NULL;
delete PNode; // звільнення пам'яті
```

Контрольні питання.

1. Чому в програмах розмір пам'яті під статичні змінні повинен бути визначений на етапі компіляції?
2. За рахунок яких ресурсів виділяється пам'ять під динамічні структури?
3. Чому динамічні структури не вимагають власного опису в програмі?
4. Як розташовуються в пам'яті динамічні величини?
5. Як здійснюється доступ до динамічних структур з програмного коду?
6. Як зв'язуються між собою елементи динамічної структури?
7. У чому основна відмінність суміжного і зв'язаного подання даних?
8. Якого типу може бути поле даних в динамічній структурі?
9. Чому для звернення до динамічної структури досить зберігати в пам'яті адресу її першого елемента?
10. За рахунок чого робота з динамічними даними уповільнює виконання програми?

Лекція 2
ОДНОСПРЯМОВАНІ І ДВОСПРЯМОВАНІ СПИСКИ

Списком називається впорядкована множина, що складається зі змінного числа елементів, до яких застосовані операції включення, виключення. Список, що відображає відносини сусідства між елементами, називається лінійним. ***Довжина списку*** дорівнює числу елементів, що містяться у списку, список нульової довжини називається порожнім списком. Списки є спосіб організації структури даних, за якої елементи деякого типу утворюють ланцюжок. Для зв'язування елементів у списку використовують систему покажчиків. У мінімальному випадку будь-який елемент лінійного списку має один покажчик, який вказує на наступний елемент у списку або є порожнім покажчиком, що інтерпретується як кінець списку.

Структура, елементами якої служать записи з одним і тим самим форматом, пов'язані один з одним за допомогою покажчиків, що зберігаються у самих елементах, називають ***пов'язаним списком***. У пов'язаному списку елементи лінійно впорядковані, але порядок визначається не номерами, як у масиві, а покажчиками, що входять до складу елементів списку. Кожен список має особливий елемент, називаний покажчиком початку списку (головою списку), який зазвичай за змістом відмінний від інших елементів. В полі покажчика останнього елемента списку знаходиться спеціальна позначка NULL, яка свідчить про кінець списку.

Лінійні пов'язані списки є найпростішими динамічними структурами даних. З усього різноманіття пов'язаних списків можна виділити наступні основні:

- односпрямовані (однозв'язані) списки;
- двоспрямовані (двозв’язані) списки;
- циклічні (кільцеві) списки.

В основному вони відрізняються видом взаємозв'язку елементів і / або допустимими операціями.

Односпрямовані (однозв'язані) списки

Найбільш простою динамічною структурою є односпрямований список, елементами якого служать об'єкти структурного типу.

Односпрямований (однозв'язний) список − це структура даних, що являє собою послідовність елементів, в кожному з яких зберігається значення і покажчик на наступний елемент списку (рис. 2.1). В останньому елементі покажчик на наступний елемент дорівнює NULL.

Опис найпростішого елемента такого списку виглядає наступним чином:

`struct` ім’я_типу {інформаційне поле; адресне поле; };

де інформаційне поле − це поле будь-якого, раніше оголошеного або стандартного типу;

адресне поле − це покажчик на об'єкт того ж типу, що і визначається структура, в нього записується адреса наступного елемента списку.

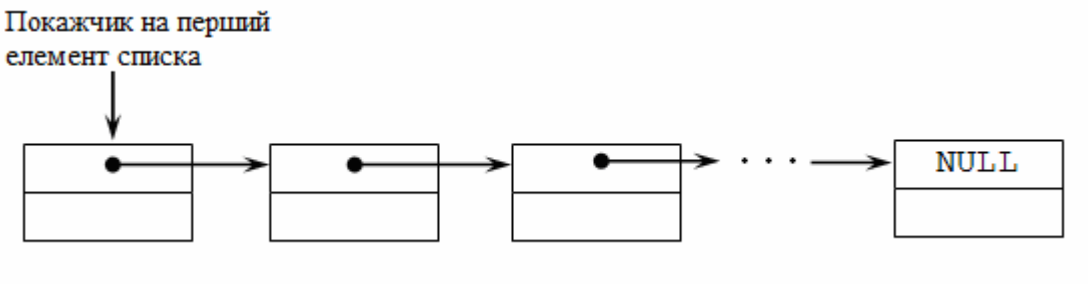


Рисунок 2.1 – Лінійні односпрямованими списки

Наприклад:

```
struct Node {
              int key;//інформаційне поле
              Node*next;//адресне поле
             };
```

Інформаційних полів може бути кілька. Наприклад:

```
struct point {
               char*name;//інформаційне поле
               int age;//інформаційне поле
               point*next;//адресне поле
              };
```

Кожен елемент списку містить ключ, який ідентифікує цей елемент. Ключ зазвичай буває або цілим числом, або рядком. Основними операціями, які здійснюються з односпрямованим списками, є:

- створення списку;
- друк (перегляд) списку;
- включення елемента у список;
- виключеня елемента зі списку;
- пошук елемента у списку;
- перевірка відсутності елементів у списку (NULL);
- видалення списку.

Особливу увагу слід звернути на те, що при виконанні будь-яких операцій з лінійним односпрямованим списком необхідно забезпечувати позиціонування будь-якого покажчика на перший елемент. В іншому випадку частина або весь список буде недоступний.

Розглянемо докладніше кожну з наведених операцій. Для опису алгоритмів цих основних операцій використовується наступне оголошення:

```
struct Single_List {//структура даних
                     int Data; //інформаційне поле
                     Single_List *Next; //адресне поле
                    };
. . . . . .
Single_List *Head; //покажчик на перший елемент списку
. . . . . .
Single_List *Current;
//покажчик на поточний элемент списку (за необхідносттю)
```

Створення односпрямованого списку

Для того, щоб створити список, потрібно створити спочатку перший елемент списку, а потім за допомогою функції додати до нього інші елементи. За відносно невеликих розмірів списку найбільш витончено і красиво є використання рекурсивної функції. Додавання може виконуватися як на початок, так і на кінець списку.

```
// створення односпрямованого списку (додавання до кінця)
void Make_Single_List(int n,Single_List** Head){
  if (n > 0) {
    (*Head) = new Single_List();//виділяємо пам'ять під новий
                                                елемент
    cout << "Введите значение ";
    cin >> (*Head)->Data;//вводимо значення інформаційного поля
    (*Head)->Next=NULL;//обнулення адресного поля
     Make_Single_List(n-1,&((*Head)->Next));
  }
}
```

Друк (перегляд) односпрямованого списку

Операція друку списку полягає в послідовному перегляді всіх елементів списку і виведення їх значень на екран. Для обробки списку створюються функція, в якій потрібно переставляти покажчик на наступний елемент списку до тих пір, поки покажчик не стане дорівнювати NULL, тобто буде досягнутий кінець списку. Реалізуємо цю функцію рекурсивно.

```
// друк односпрямованого списку
void Print_Single_List(Single_List* Head) {
  if (Head != NULL) {
    cout << Head->Data << "\t";
    Print_Single_List(Head->Next);
    //перехід до наступного елемента
  }
  else cout << "\n";
}
```

Включення елемента до односпрямованого списку

У динамічні структури легко додавати елементи, тому що для цього достатньо змінити значення адресних полів. Включення першого і наступних елементів списку відрізняються один від одного. Тому у функції, що реалізує дану операцію, спочатку здійснюється перевірка, на яке місце вставляється елемент. Далі реалізується відповідний алгоритм додавання (рис. 2.2).

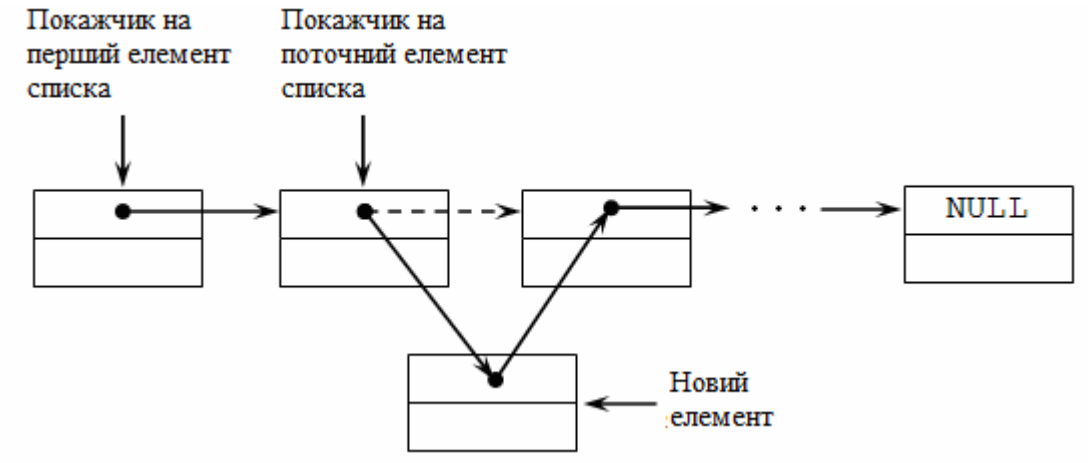


Рисунок 2.2 – Включення елемента в односпрямований список

```
/*включення елемента з заданим номером до односпрямованого списку*/
Single_List* Insert_Item_Single_List(Single_List* Head,
                        Int Number, int DataItem) {
    Number--;
    Single_List* NewItem=new(Single_List);
    NewItem->Data=DataItem;
    NewItem->Next=NULL;
    if (Head==NULL) {//список порожній
       Head=NewItem;//створюємо перший елемент списку
     }
    else {//список не порожній
       Single_List* Current=Head;
       for(int i=1; i<Number && Current->Next!=NULL; i++)
             Current=Current->Next;
        if (Number==0) {
//вставляємо новий елемент на перше місто
           NewItem->Next=Head;
           Head=NewItem;
              }
        else {//вставляємо новий елемент на не перше місто
                     if (Current->Next!=NULL)
                           NewItem->Next=Current->Next;
                           Current->Next=NewItem;
            }
   }
return Head;
}
```

Виключеня елемента з односпрямованого списку

З динамічних структур можна виключати елементи, тому що для цього достатньо змінити значення адресних полів. Операція виключення елемента

односпрямованого списку здійснює виключення елемента, на який встановлено покажчик поточного елемента. Після виключення покажчик поточного елемента встановлюється на попередній елемент списку або на новий початок списку, якщо видаляється перший.

Алгоритми виключення першого і наступних елементів списку відрізняються один від одного. Тому у функції, що реалізує дану операцію, здійснюється перевірка, який об'єкт був видалений. Далі реалізується відповідний алгоритм виключення (рис. 2.3).

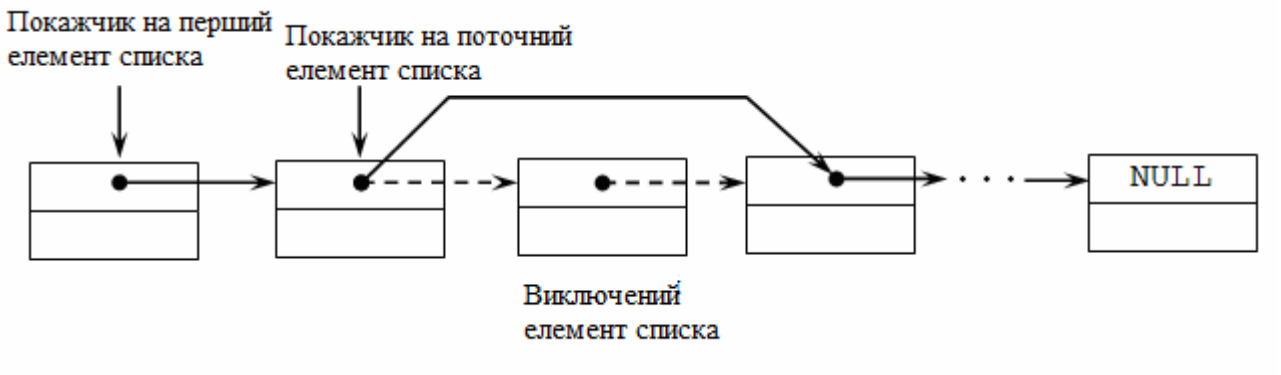


Рисунок 2.3 – Виключення елемента з односпрямованого списку

```
//виключення елемента за заданим номером з односпрямованому списку
Single_List* Delete_Item_Single_List(Single_List* Head,
      int Number){
  Single_List *ptr;//допоміжний покажчик
  Single_List *Current = Head;
  for (int i = 1; i < Number && Current != NULL; i++)
    Current = Current->Next;
  if (Current != NULL){//перевірка на коректність
    if (Current == Head){//виключення першого элемента
      Head = Head->Next;
      delete(Current);
      Current = Head;
    }
    else {//виключення не першого елемента
      ptr = Head;
      while (ptr->Next != Current)
        ptr = ptr->Next;
      ptr->Next = Current->Next;
      delete(Current);
      Current=ptr;
    }
  }
  return Head;
}
```

Пошук елемента в односпрямованому списку

Операція пошуку елемента у списку полягає в послідовному перегляді всіх елементів списку до тих пір, поки поточний елемент не буде містити задане значення або поки не буде досягнутий кінець списку. В останньому випадку фіксується відсутність шуканого елемента у списку (функція приймає значення false).

```
// пошук елемента в односпрямованому списку
bool Find_Item_Single_List(Single_List* Head, int
DataItem){
  Single_List *ptr; //допоміжний покажчик
  ptr = Head;
  while (ptr != NULL){//доки не кінець списку
    if (DataItem == ptr->Data) return true;
    else ptr = ptr->Next;
  }
  return false;
}
```

Видалення односпрямованого списку

Операція видалення списку полягає у звільненні динамічної пам'яті. Для даної операції створюється функція, в якій потрібно переставляти покажчик на наступний елемент списку до тих пір, поки покажчик не буде дорівнювати NULL, тобто не буде досягнутий кінець списку. Реалізуємо рекурсивну функцію.

```
// Звільнення пам'яті, виділеної під односпрямований список
void Delete_Single_List(Single_List* Head){
  if (Head != NULL){
    Delete_Single_List(Head->Next);
    delete Head;
  }
}
```

Таким чином, односпрямований список має тільки один покажчик в кожному елементі. Це дозволяє мінімізувати витрату пам'яті на організацію такого списку. Одночасно це дозволяє здійснювати переходи між елементами лише в одному напрямку, що часто збільшує час, який витрачається на обробку списку. Наприклад, для переходу до попереднього елемента необхідно здійснити перегляд списку з початку до елемента, покажчик якої встановлено на поточний елемент.

Двоспрямовані (двозв’язані) списки

Для прискорення багатьох операцій доцільно застосовувати переходи між елементами списку в обох напрямках. Це реалізується за допомогою двоспрямованих списків, які є складною динамічною структурою.

Двоспрямовані (двозв’язані) списку − це структура даних, що складається з послідовності елементів, кожен з яких містить інформаційну частину і два

покажчика на сусідні елементи (рис. 2.4). При цьому два сусідні елементи повинні містити взаємні посилання один на одного.

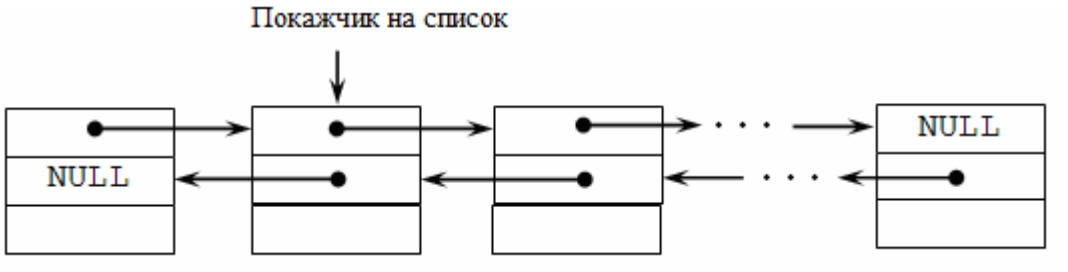


Рисунок 2.4 − Двоспрямований (двозв’язаний) список

Опис найпростішого елемента такого списку виглядає наступним чином:

```
struct ім'я_типу {
                    інформаційне поле;
                    адресне поле 1;
                    адресне поле 2;
                  };
```

де інформаційне поле − це поле будь-якого, раніше оголошеного або стандартного, типу;
адресне поле 1 – це покажчик на об'єкт того самого типу, що і визначена структура, в нього записується адреса наступного елемента списку;
адресне поле 2 - це покажчик на об'єкт того самого типу, що і визначена структура, в нього записується адреса попереднього елемента списку.
Наприклад:

```
struct list {
               type elem ;
               list *next, *prev ;
             }
list *headlist ;
```

де `type` - тип інформаційного поля елемента списку;
`* Next, * prev` - покажчики на наступний і попередній елементи цієї структури відповідно. Змінна-покажчик `headlist` задає список як єдиний програмний об'єкт, її значення – покажчик на перший (або титульний) елемент списку.

Основні операції, що виконуються над двоспрямованим списком, ті самі, що й для односпрямованого списку. Тому що двоспрямований список більш гнучкий, ніж односпрямований, то при включенні елемента у список, потрібно використовувати покажчик як на елемент, за яким відбувається включення, так і покажчик на елемент, перед яким відбувається включення. При виключення елемента зі списку потрібно використовувати як покажчик на сам виключається елемент, так і покажчики на попередній або наступний за виключається елементи. Але тому що елемент двоспрямованого списку має два покажчика, то при виконанні операцій включення / виключення елемента треба змінювати

більше зв'язків, ніж в односпрямованому списку. Розглянемо основні операції, здійснювані з двоспрямованими списками, такі як:

- створення списку;
- друк (перегляд) списку;
- вставка елемента у список;
- видалення елемента зі списку;
- пошук елемента у списку;
- перевірка порожнини списку;
- видалення списку.

Особливу увагу слід звернути на те, що на відміну від односпрямованого списку тут немає необхідності забезпечувати позиціонування будь-якого покажчика саме на перший елемент списку, тому що завдяки двом вказівникам в елементах можна отримати доступ до будь-якого елементу списку з будь-якого іншого елемента, здійснюючи переходи в прямому або зворотному напрямку. Однак за правилами хорошого тону програмування покажчик бажано ставити на заголовок списку.

Для опису алгоритмів цих основних операцій використовується наступне оголошення:

```
struct Double_List {//структура даних
                     int Data; //інформаційне поле
                     Double_List *Next, //адресне поле
                                 *Prior; //адресне поле
                    };
. . . . . .
Double_List *Head; //покажчик на перший елемент списку
. . . . . .
Double_List *Current;
//покажчик на поточний елемент списку (за необхідності)
```

Створення двоспрямованого списку

Для того, щоб створити список, потрібно створити спочатку перший елемент списку, а потім за допомогою функції додати до нього інші елементи. Додавання може виконуватися як на початок, так і на кінець списку. Реалізуємо рекурсивную функцію.

```
// створення двоспрямованого списку (додавання до кінця)
void Make_Double_List(int n,Double_List** Head,
          Double_List* Prior){
  if (n > 0) {
    (*Head) = new Double_List();
     //виділяємо пам’ять під новий елемент
    cout << "Введіть значення";
    cin >> (*Head)->Data;
    //вводимо значення інформаційного поля
    (*Head)->Prior = Prior;
```

```
    (*Head)->Next=NULL;//обнулення адресного поля
    Make_Double_List(n-1,&((*Head)->Next),(*Head));
  }
  else (*Head) = NULL;
}
```

Друк (перегляд) двоспрямованого списку

Операція друку списку для двоспрямованого списку реалізується абсолютно аналогічно відповідної функції для односпрямованого списку. Переглядати двоспрямований список можна в обох напрямках.

```
// друк двоспрямованого списку
void Print_Double_List(Double_List* Head) {
  if (Head != NULL) {
    cout << Head->Data << "\t";
    Print_Double_List(Head->Next);
    //перехід до наступного елемента
  }
  else cout << "\n";
}
```

Включення елемента в двоспрямований список

У динамічних структур легко додавати елементи, тому що для цього достатньо змінити значення адресних полів. Операція включення реалізовується аналогічно функції включення для односпрямованого списку, тільки з урахуванням особливостей двоспрямованого списку (рис. 2.5).

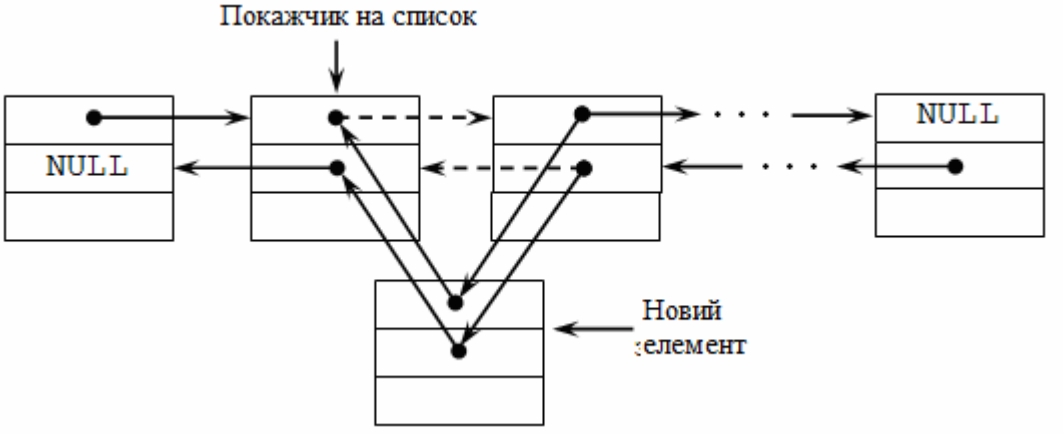


Рисунок 2.5 – Включення елемента в двоспрямований список

```
//включення елемента з заданим номером до двоспрямованого списку
Double_List* Insert_Item_Double_List(Double_List* Head,
      int Number, int DataItem){
  Number--;
  Double_List *NewItem=new(Double_List);
  NewItem->Data=DataItem;
  NewItem->Prior=NULL;
```

```
  NewItem->Next = NULL;
  if (Head == NULL) {//список порожній
    Head = NewItem;
  }
  else {//список не порожній
    Double_List *Current=Head;
    for(int i=1; i < Number && Current->Next!=NULL; i++)
    Current=Current->Next;
    if (Number == 0){
    //вставляємо новий елемент на перше місто
      NewItem->Next = Head;
      Head->Prior = NewItem;
      Head = NewItem;
    }
    else {//вставляємо новий елемент на не перше місто
      if (Current->Next != NULL) Current->Next->Prior =
NewItem;
      NewItem->Next = Current->Next;
      Current->Next = NewItem;
           NewItem->Prior = Current;
      Current = NewItem;
    }
  }
  return Head;
}
```

Виключення елемента з двоспрямованого списку

З динамічних структур можна виключати елементи, тому що для цього достатньо змінити значення адресних полів. Операція виключення елемента з двоспрямованого списку здійснюється багато в чому аналогічно виключення з односпрямованого списку (рис. 2.6).

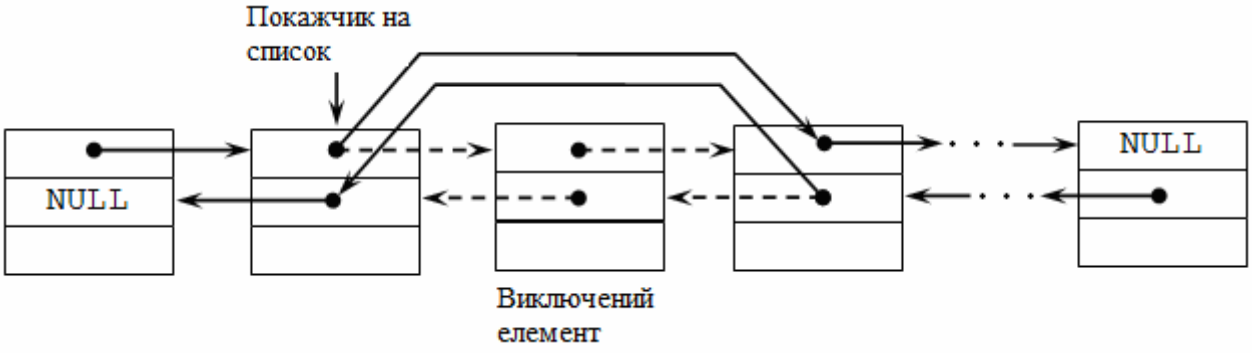


Рисунок 2.6 – Виключення елемента з двоспрямованого списку

```
/*виключення елемента з заданим номером з двоспрямованого списка*/
Double_List* Delete_Item_Double_List(Double_List* Head,
      int Number){
```

```
  Double_List *ptr;//допоміжний покажчик
  Double_List *Current = Head;
  for (int i = 1; i < Number && Current != NULL; i++)
    Current = Current->Next;
  if (Current != NULL){//перевірка на коректність
    if (Current->Prior == NULL){//виключення першого елемента
      Head = Head->Next;
      delete(Current);
      Head->Prior = NULL;
      Current = Head;
    }
    else {//виключення не першого елемента
      if (Current->Next == NULL) {
      //виключаємо останній елемент
        Current->Prior->Next = NULL;
        delete(Current);
        Current = Head;
      }
      else {//виключення не першого та не останнього елемента
        ptr = Current->Next;
        Current->Prior->Next =Current->Next;
        Current->Next->Prior =Current->Prior;
        delete(Current);
        Current = ptr;
      }
    }
  }
 return Head;
  }
```

Пошук елемента в двоспрямованому списку

Операція пошуку елемента в двоспрямованому списку реалізується абсолютно аналогічно відповідній функції для односпрямованого списку.

Пошук елемента в двоспрямованому списку можна вести:

а) переглядаючи елементи від початку до кінця списку;

б) переглядаючи елементи від кінця списку до початку;

в) переглядаючи список в обох напрямках одночасно: від початку до середини списку і від кінця до середини (враховуючи, що елементів у списку може бути парна або непарна кількість).

```
// пошук елемента в двоспрямованому списку
bool Find_Item_Double_List(Double_List* Head,
    int DataItem){
  Double_List *ptr; //допоміжний покажчик
  ptr = Head;
  while (ptr != NULL){//доки не кінець списку
    if (DataItem == ptr->Data) return true;
```

```
    else ptr = ptr->Next;
    }
  return false;
}
```

Перевірка порожнини двоспрямованого списку

Операція перевірки двоспрямованого списку на порожнину здійснюється аналогічно перевірці односпрямованого списку.

```
// перевірка порожнини двоспрямованого списку
bool Empty_Double_List(Double_List* Head){
  if (Head!=NULL)  return false;
  else  return true;
}
```

Видалення двоспрямованого списку

Операція видалення двоспрямованого списку реалізується аналогічно видалення односпрямованого списку.

```
// звільнення пам'яті, виділеної під двоспрямований список
void Delete_Double_List(Double_List* Head){
  if (Head != NULL){
    Delete_Double_List(Head->Next);
    delete Head;
  }
}
```

Приклад 1. N-натуральних чисел є елементами двоспрямованого списку L, обчислити: $X_1 * X_n + X_2 * X_{n-1} + ... + X_n * X_1$. Вивести на екран кожний добуток та підсумкову суму.

Алгоритм:

1. Створюємо структуру.
2. Формуємо список цілих чисел.
3. Просуваємося за списком: від початку до кінця і від кінця до початку в одному циклі, перемножуємо дані, що містяться у відповідних елементах списку.
4. Підсумовуємо отримані результати.
5. Виводимо на друк.

Створення структури, формування списку і вивід на друк розглянуті раніше. Надамо функції для реалізації просування за списком в обох напрямках і знаходження підсумкової суми.

```
    // пошук останнього елемента
Double_List* Find_End_Item_Double_List(Double_List*
Head){
  Double_List *ptr; //допоміжний покажчик
  ptr = Head;
  while (ptr->Next != NULL){
```

```
    ptr = ptr->Next;
  }
  return ptr;
}

//підсумкова сума добутків
void Total_Sum(Double_List* Head) {
  Double_List* lel = Head;
  Double_List* mel = Find_End_Item_Double_List(Head);
  int mltp,sum=0;
  while(lel != NULL) {
    mltp = (lel->Data)*(mel->Data);//множення елементів
    printf("\n\n%d * %d = %d",lel->Data,mel->Data,mltp);
    sum = sum + mltp;//додавання добутків
    lel = lel->Next;
    //йдемо по списку від першого елемента до останнього
    mel = mel->Prior;
    //йдемо за списком від останнього елемента до першого
  }
  printf("\n\n Итоговая сумма равна %d",sum);
}
```

Контрольні питання

1. Чи кожен список є зв'язаним? Обгрунтуйте відповідь.
2. У чому відмінність першого елемента односпрямованого (двоспрямованого) списку від інших елементів цього ж списку?
3. У чому відмінність останнього елемента односпрямованого (двоспрямованого) списку від інших елементів цього ж списку?
4. Чому при роботі з односпрямованим списком необхідне позиціонування на перший елемент списку?
5. Чому при роботі з двоспрямованим списком не обов'язкове позиціонування на перший елемент списку?
6. У чому принципові відмінності виконання додавання (видалення) елемента на першу і будь-яку іншу позиції в односпрямованому списку?
7. У чому принципові відмінності виконання основних операцій в односпрямованих і двоспрямованих списках?
8. З якою метою в програмах виконується перевірка на порожнину односпрямованого (двоспрямованого) списку?
9. З якою метою в програмах виконується видалення односпрямованого (двоспрямованого) списку після закінчення роботи з ним? Як зміниться робота програми, якщо операцію видалення списку не виконувати?

Лекція 3
ЧЕРГА ТА СТЕК

У списках доступ до елементів відбувається за допомогою адресації, при цьому доступ до окремих елементів не обмежений. Але існують також і такі спискові структури даних, в яких є обмеження доступу до елементів. Одним з представників таких спискових структур є стековий список або просто стек.

Стек (англ. Stack – стопка) – це структура даних, в якій новий елемент завжди записується в її початок (вершину) і черговий елемент читається також завжди вибирається з її початку (рис. 3.1). У стеках використовується метод доступу до елементів LIFO (Last Input − First Output, "останнім прийшов − першим вийшов"). Найчастіше принцип роботи стека порівнюють зі стопкою тарілок: щоб взяти другу зверху, потрібно спочатку взяти верхню.

Стек − це список, у якого доступний один елемент (одна позиція). Цей елемент називається вершиною стека. Взяти елемент можна тільки з вершини стека, додати елемент можна тільки у вершину стека. Наприклад, якщо записані у стек числа 1, 2, 3, то при подальшому витяганні отримаємо 3, 2, 1.

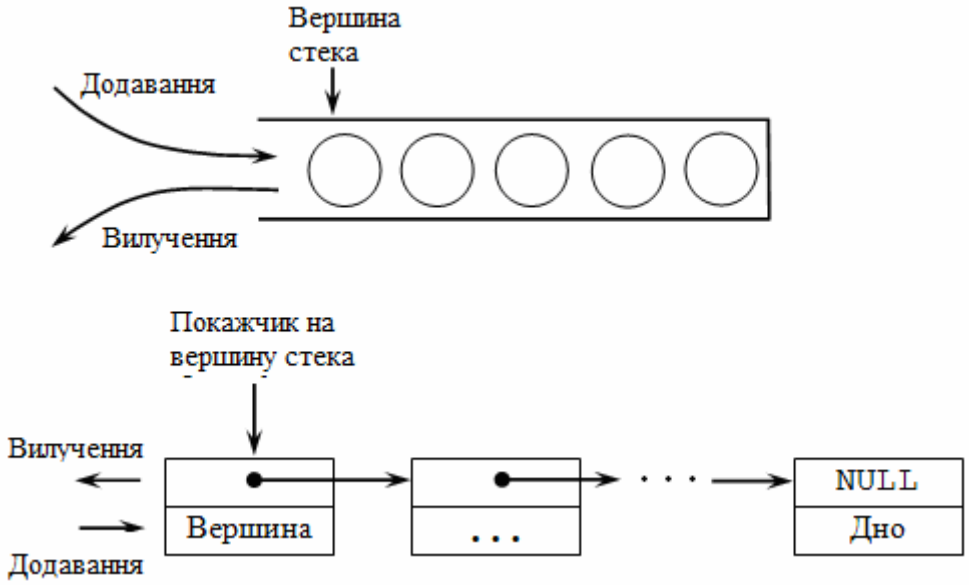


Рисунок 3.1 − Стек та його організація

Опис стека виглядає наступним чином:

```
struct ім'я_типу {
                    інформаційне поле;
                    адресне поле;
                  };
```

де інформаційне поле − це поле будь-якого раніше оголошеного або стандартного типу; адресне поле − це покажчик на об'єкт того ж типу, що і структура, в нього записується адреса наступного елемента стека.
Наприклад:

```
struct list {
             type  pole1;
```

```
            list *pole2;
           } stack;
```

Стек як динамічну структуру даних легко створити на основі лінійного списку. Оскільки робота завжди йде з заголовком стека, тобто не потрібно здійснювати перегляд елементів, видалення і вставку елементів у середину або кінець списку, то досить використовувати економічний по пам'яті лінійний односпрямований список. Для такого списку досить зберігати покажчик вершини стека, який вказує на перший елемент списку. Якщо стек порожній, то списку не існує, і покажчик набуває значення NULL.

Оголошення елементів стека аналогічно оголошенню елементів лінійного односпрямованого списку. Тому оголосимо стек через оголошення лінійного односпрямованого списку:

```
struct Stack {
              Single_List *Top;//вершина стека
             };
. . . . . .
  Stack *Top_Stack;//покажчик на вершину стека
```

Основні операції, які застосовуються до стека:

- створення стека;
- друк (перегляд) стека;
- додавання елемента у вершину стека;
- витяг елемента з вершини стека;
- перевірка порожнини стека;
- очищення стека.

Реалізацію цих операцій розглянемо у вигляді відповідних функцій, які, у свою чергу, використовують функції операцій з лінійним односпрямованим списком. Звернемо увагу, що у функції створення стека використовується функція додавання елемента у вершину стека.

```
// створення стека
void Make_Stack(int n, Stack* Top_Stack){
  if (n > 0) {
    int tmp;//допоміжна змінна
    cout << "Введите значение ";
    cin >> tmp; //ввести значення інформаційного поля
    Push_Stack(tmp, Top_Stack);
    Make_Stack(n-1,Top_Stack);
  }
}

//друк стека
void Print_Stack(Stack* Top_Stack){
  Print_Single_List(Top_Stack->Top);
}
```

```
//додавання елемента до вершини стека
void Push_Stack(int NewElem, Stack* Top_Stack){
  Top_Stack->Top =Insert_Item_Single_List(Top_Stack-
>Top,1,NewElem);
}
//вилучення елемента з вершини стека
int Pop_Stack(Stack* Top_Stack){
  int NewElem = NULL;
  if (Top_Stack->Top != NULL) {
    NewElem = Top_Stack->Top->Data;
    Top_Stack->Top = Delete_Item_Single_List(Top_Stack-
>Top,0);
//видалення вершини
  }
  return NewElem;
}
//перевірка порожнини стека
bool Empty_Stack(Stack* Top_Stack){
  return Empty_Single_List(Top_Stack->Top);
}
//очищення стека
void Clear_Stack(Stack* Top_Stack){
  Delete_Single_List(Top_Stack->Top);
}
```

Приклад 1. Дано рядок символів. Перевірте правильність розставлення в ньому круглих дужок.

При розв'язку цього завдання будемо використовувати стек. Надамо головну функцію та функцію для перевірки правильності розстановлення круглих дужок.

```
// головна функція
int _tmain(int argc, _TCHAR* argv[]){
  char text[255];
  printf("Введіть текст, який містить\"(\" и \")\" \n");
  gets(text);
  Check_Brackets (text);
  system("pause");
  return 0;
}

//функція перевірки правильності розстановлення дужок
void Check_Brackets (char *text){
  int i;
  int flag=1;
  Stack *Top_Stack;
```

```
Top_Stack = new Stack();
for(i=0;i<strlen(text); i++) {
  if(text[i]==')' ) {
    if(Empty_Stack(Top_Stack)) {
// Спроба видалити нульовий елемент стека
       flag=0;
       break;
     }
 if(Top_Stack->Top->Data == '(')
      Pop_Stack(Top_Stack);
   else {
      flag=0;
      break;
     }
   }
   if(text[i]=='(')
   Push_Stack(text[i],Top_Stack);
  }
  if(flag!=0 && Empty_Stack(Top_Stack))
   printf("Вірно!");
  else printf("Невірно!");
  Clear_Stack(Top_Stack);
  printf("\n");
 }
```

Черги

Черга − це структура даних, що являє собою послідовність елементів, створена в порядку їх надходження. Кожен новий елемент розміщується в кінці черги; елемент, що стоїть на початку черги, вибирається з неї першим. У черзі використовується принцип доступу до елементів FIFO (First Input − First Output, "перший прийшов − першим вийшов") (рис. 3.2). У черзі доступні два елементи (дві позиції): початок черги і кінець черги. Помістити елемент можна тільки в кінець черги, а взяти елемент тільки з її початку. Прикладом може бути звичайна черга в магазині.

Оголошення черги виглядає наступним чином:

```
struct ім'я_типу {
                    інформаційне поле;
                    адресне поле1;
                    адресне поле2;
                  };
```

де інформаційне поле – це поле будь-якого, раніше оголошеного або стандартного, типу; адресне поле1, адресне поле 2 – це покажчики на об'єкти того ж типу, за яким визначається структура, в них записуються адреси першого і наступного елементів черги.

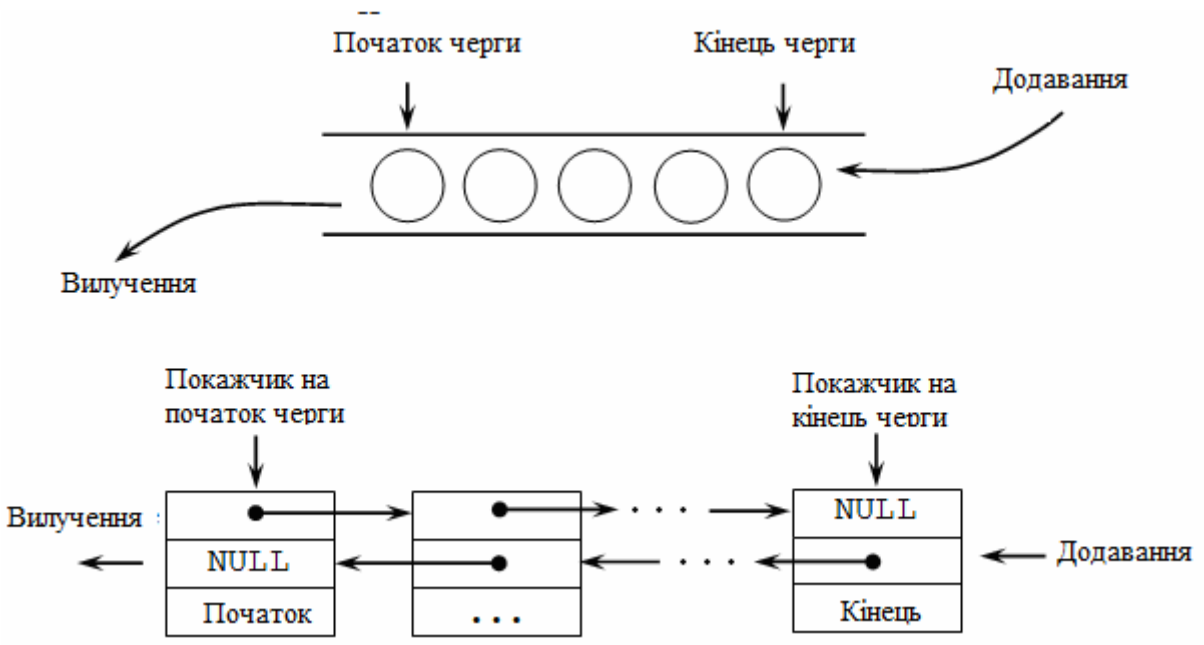


Рисунок 3.2 – Черга та її створення

Наприклад:

1 спосіб: адресне поле посилається на оголошену структуру.

```
struct list2 {
              type  pole1;
              list2 *pole1, *pole2;
             }
```

2 спосіб: адресне поле посилається на раніше оголошену структуру.

```
struct list1 {
              type  pole1;
              list1  *pole2;
             }
struct ch3 {
            list1 *beg, *next ;
           }
```

Чергу як динамічну структуру даних легко створити на основі лінійного списку. Оскільки робота відбувається з обома кінцями черги, то переважно використовувати лінійний двоспрямований список. Хоча для роботи з таким списком досить мати один покажчик на будь-який елемент списку, тут доцільно зберігати два покажчику – один на початок списку (звідки вилучаємо елементи) і один на кінець списку (куди додаємо елементи). Якщо черга порожня, то списку не існує, і покажчики набувають значення NULL.

Опис елементів черги аналогічно опису елементів лінійного двоспрямованного списку. Тому оголосимо чергу через оголошення лінійного двоспрямованного списку:

```
struct Queue {
              Double_List *Begin;//початок черги
              Double_List *End; //кінець черги
             };
```

```
. . . . . .
Queue *My_Queue;//покажчик на чергу
```
Основні операції, які виконуються з чергою:

- створення черги;
- друк (перегляд) черги;
- додавання елемента в кінець черги;
- витяг елемента з початку черги;
- перевірка порожнечі черзі;
- очищення черги.

Реалізацію цих операцій надамо у вигляді відповідних функцій, які, у свою чергу, використовують функції операцій з лінійним двоспрямованим списком.

```
//створення черги
void Make_Queue(int n, Queue* End_Queue){
  Make_Double_List(n,&(End_Queue->Begin),NULL);
  Double_List *ptr; //допоміжний покажчик
  ptr = End_Queue->Begin;
  while (ptr->Next != NULL)
    ptr = ptr->Next;
  End_Queue->End = ptr;
}

//друк черги
void Print_Queue(Queue* Begin_Queue){
  Print_Double_List(Begin_Queue->Begin);
}

//додавання елемента на кінець черги
void Add_Item_Queue(int NewElem, Queue* End_Queue){
  End_Queue->End = Insert_Item_Double_List(End_Queue-
>End,
     0, NewElem)->Next;
}

//вилучення елемента з початку черги
int Extract_Item_Queue(Queue* Begin_Queue){
  int NewElem = NULL;
  if (Begin_Queue->Begin != NULL) {
   NewElem = Begin_Queue->Begin->Data;
   Begin_Queue->Begin=
Delete_Item_Double_List(Begin_Queue->Begin,0);
//видаляємо вершину
  }
  return NewElem;
}
```

//перевірка порожнини черги
```
bool Empty_Queue(Queue* Begin_Queue){
  return Empty_Double_List(Begin_Queue->Begin);
}
```

//очистка черги
```
void Clear_Queue(Queue* Begin_Queue){
  return Delete_Double_List(Begin_Queue->Begin);
}
```

Приклад 2. Дана послідовність ненульових цілих чисел. Ознакою кінця послідовності є число 0. Знайдіть серед них перший найбільший від'ємний елемент. Якщо такого елемента немає, то виведіть повідомлення про це. У цьому завданні будемо використовувати основні операції для роботи з чергою, розглянуті раніше. Надамо головну функцію та функцію для реалізації пошуку першого найбільшого від'ємного елемента.

//головна функція
```
int _tmain(int argc, _TCHAR* argv[]){
  int n;
  Queue *My_Queue;
  My_Queue = new Queue();
  Make_Queue(1,My_Queue);
  while (My_Queue->End->Data != 0){
    cout << "Введіть значення ";
    cin >> n;
    Add_Item_Queue(n,My_Queue);
  }
  cout << "\nЧерга: \n";
  Print_Queue(My_Queue);
  Find_Max_Negative_Element(My_Queue);
  system("pause");
  return 0;
}
```

//функція пошуку першого найбільшого від'ємного елемента
```
void Find_Max_Negative_Element(Queue* Begin_Queue){
  int tmp;
  int max=Extract_Item_Queue(Begin_Queue);
  while (Begin_Queue->Begin->Data != 0) {
    tmp = Extract_Item_Queue(Begin_Queue);
    if (max > 0 || tmp < 0 && abs(tmp) < abs(max))
      max = tmp;
  }
  if (max > 0) printf("Елементів немає!");
  else printf("Є такий елемент: %d", max);
}
```

Циклічні (кільцеві) списки

Циклічний (кільцевий) список − це структура даних, що являє собою послідовність елементів, останній елемент якої містить покажчик на перший елемент списку, а перший (у разі двоспрямованого списку) – на останній. Основна особливість такого створення полягає в тому, що в цьому списку немає елементів, які містять порожні покажчики, і, отже, не можна виділити крайні елементи.

Циклічні списки, так само як і лінійні, бувають односпрямованими і двонаправленими. Циклічний односпрямований список схожий на лінійний односпрямований список, але його останній елемент містить покажчик, що зв'язує його з першим елементом (рис. 3.3).

Для повного обходу такого списку досить мати покажчик на довільний елемент, а не на перший, як у лінійному односпрямованому списку. Поняття "першого" елемента тут досить умовно і не завжди потрібно. Хоча іноді буває корисно виділити певний елемент як "перший" шляхом установлення на ньому спеціального покажчика. Це потрібно, наприклад, для запобігання "зациклення" при перегляді списку.

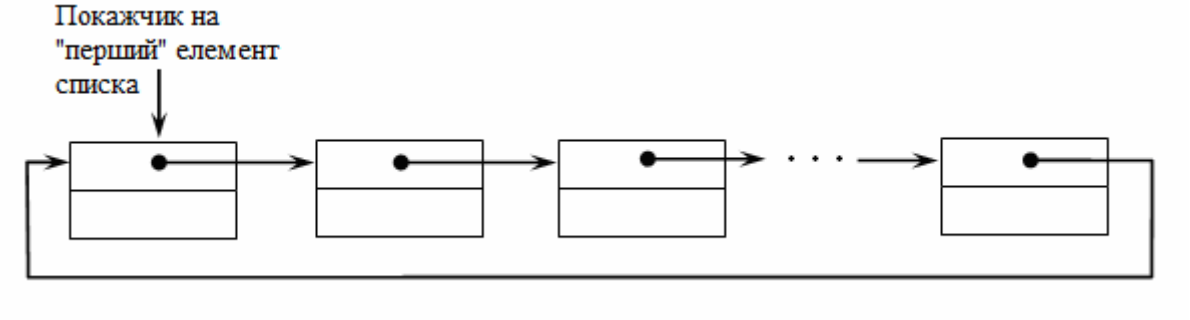


Рисунок 3.3 − Циклічний односпрямований список

Основні операції, які здійснюютьсяі з циклічним односпрямованим списком:

- створення списку;
- друк (перегляд) списку;
- вставка елемента в список;
- видалення елемента зі списку;
- пошук елемента у списку;
- перевірка порожнини списку;
- видалення списку.

Для опису алгоритмів цих основних операцій будемо використовувати ті самі оголошення, що і для лінійного односпрямованого списку.

Надамо функції перерахованих основних операцій при роботі з циклічним односпрямованим списком.

```
// створення циклічного односпрямованого списку
void Make_Circle_Single_List(int n,
       Circle_Single_List** Head,Circle_Single_List*
Loop){
    if (n > 0) {
```

```
        (*Head) = new Circle_Single_List();
       //видалення пам'яті під новий елемент
        if (Loop == NULL) Loop = (*Head);
        cout << "Введіть значення ";
        cin >> (*Head)->Data;
        //вводимо значення інформаційного поля
        (*Head)->Next=NULL;//обнулення адресного поля
        Make_Circle_Single_List(n-1,&((*Head)->
Next),Loop);
    }
    else {
        (*Head) = Loop;
    }
}

//друк циклічного односпрямованого списку
void Print_Circle_Single_List(Circle_Single_List* Head) {
    Circle_Single_List* ptr=Head;
    //допоміжний покажчик
    do {
       cout << ptr->Data << "\t";
       ptr=ptr->Next;
    } while (ptr!=Head);
    cout << "\n";
}

/*включення елемента після заданого номера у циклічний
односпрямований список*/
Circle_Single_List*
Insert_Item_Circle_Single_List(Circle_Single_List* Head,
      int Number, int DataItem){
  Circle_Single_List *Current = Head;
  //стати на перший елемент
  Circle_Single_List *NewItem = new(Circle_Single_List);
  //створення нового елемента
  NewItem->Data = DataItem;
  if (Head == NULL) {//список порожній
    NewItem->Next = NewItem;
    Head = NewItem;
  }
  else {//список непорожній
    for (int i = 1; i < Number; i++)
      Current = Current->Next;
    NewItem->Next = Current->Next;
    Current->Next = NewItem;
```

```
  }
  return Head;
}
```

/*видалення елемента з заданим номером з циклічного односпрямованого списку*/

```
Circle_Single_List* Delete_Item_Circle_Single_List
      (Circle_Single_List* Head, int Number){
  if (Head != NULL){
    Circle_Single_List *Current = Head;
    if (Head->Next != Head){
      for (int i = 1; i < Number; i++)
        Current = Current->Next;
        Circle_Single_List *ptr = Head;
      while (ptr->Next != Current)
        ptr = ptr->Next;
      // безпосереднє виключення елемента
      ptr->Next = Current->Next;
      if (Head = Current) Head = Current->Next;
      delete(Current);
    }
    else{
      Head = NULL;
      delete(Current);
    }
  }
  return Head;
}
```

//пошук елемента у циклічному односпрямованом списку

```
bool Find_Item_Circle_Single_List(Circle_Single_List*
Head,
         int DataItem){
  Circle_Single_List *ptr = Head;
  //допоміжний покажчик
  do {
    if (DataItem == ptr->Data) return true;
    else ptr = ptr->Next;
    }
  while (ptr != Head);
  return false;
}
```

//перевірка порожнини циклічного односпрямованого списку

```
bool Empty_Circle_Single_List(Circle_Single_List* Head){
```

```
  return (Head != NULL ? false : true);
}

//видалення циклічного односпрямованого списка
void Delete_Circle_Single_List(Circle_Single_List* Head){
  if (Head != NULL){
    Head = Delete_Item_Circle_Single_List(Head, 1);
    Delete_Circle_Single_List(Head);
  }
}
```

Циклічний двоспрямований список схожий на лінійний двоспрямований список, але його будь-який елемент має два покажчика, один з яких вказує на наступний елемент у списку, а другий − на попередній елемент (рис. 3.4).

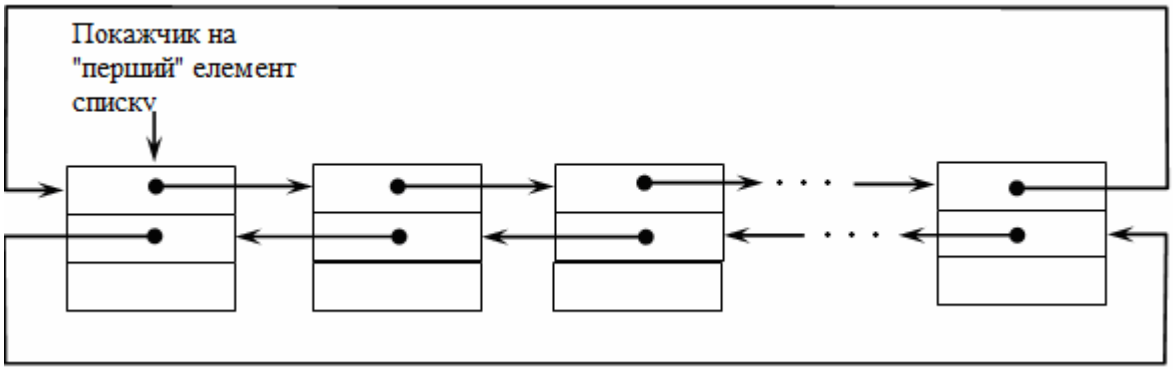


Рисунок 3.4 − Циклічний двоспрямований список

Основні операції, які здійснюються з циклічним двоспрмованим списком:

- створення списку;
- друк (перегляд) списку;
- вставка елемента у список;
- видалення елемента зі списку;
- пошук елемента у списку;
- перевірка порожнини списку;
- видалення списку.

Для опису алгоритмів цих основних операцій будемо використовувати ті самі оголошення, що і для лінійного двоспрямованого списку. Надамо функції перерахованих основних операцій при роботі з циклічним двоспрямованим списком.

```
// створення циклічного двоспрямованого списку
Circle_Double_List* Make_Circle_Double_List(int n,
      Circle_Double_List** Head,Circle_Double_List*
Loop){
  Circle_Double_List* ptr;//допоміжний покажчик
  if (n > 0) {
    (*Head) = new Circle_Double_List();
```

```
    //видалення пам'яті під новий елемент
    if (Loop == NULL) Loop = (*Head);
      cout << "Введите значение ";
    cin >> (*Head)->Data;
    //вводимо значення інформаційного поля
    (*Head)->Next=NULL;//обнулення адресного поля
    ptr = Make_Circle_Double_List(n-1,&((*Head)-
>Next),Loop);
    if ((*Head)->Next != NULL)
      (*Head)->Next->Prior = (*Head);
    if ((*Head)->Prior == NULL)
      (*Head)->Prior = ptr;
    if (ptr == NULL)
      return *Head;
    else return ptr;
    }
    else {
      (*Head) = Loop;
    return NULL;
  }
}

//друк циклічного двоспрямованого списку
void Print_Circle_Double_List(Circle_Double_List* Head) {
    Circle_Double_List* ptr=Head;
    //допоміжний покажчик
    do {
       cout << ptr->Data << "\t";
       ptr=ptr->Next;
    } while (ptr!=Head);
    cout << "\n";
}

/*включення елемента після заданого номера у циклічний двоспрямований
список*/
Circle_Double_List* Insert_Item_Circle_Double_List
   (Circle_Double_List* Head, int Number, int DataItem){
  Circle_Double_List *Current = Head;
  //встали на перший елемент
  Circle_Double_List *NewItem = new(Circle_Double_List);
  //створення нового елемента
  NewItem->Data = DataItem;
  if (Head == NULL) {//список порожній
    NewItem->Next = NewItem;
    NewItem->Prior = NewItem;
```

```
    Head = NewItem;
  }
  else {//список не порожній
    for (int i = 1; i < Number; i++)
      Current = Current->Next;
    NewItem->Next = Current->Next;
    Current->Next = NewItem;
    NewItem->Prior = Current;
    NewItem->Next->Prior = NewItem;
  }
  return Head;
}
```

/*видалення елемента з заданим номером з циклічного двоспрямованого списку*/

```
Circle_Double_List*
Delete_Item_Circle_Double_List(Circle_Double_List* Head,
      int Number){
  if (Head != NULL){
    Circle_Double_List *Current = Head;
    if (Head->Next != Head){
      for (int i = 1; i < Number; i++)
        Current = Current->Next;
      Circle_Double_List *ptr = Current->Next;
      Current->Prior->Next = Current->Next;
      Current->Next->Prior = Current->Prior;
      if (Head = Current) //виключення першого елемента
        Head = Current->Next;
      delete(Current);
    }
    else{
      Head = NULL;
      delete(Current);
    }
  }
  return Head;
}
```

//пошук елемента у циклічному двоспрямованому списку

```
bool Find_Item_Circle_Double_List(Circle_Double_List*
Head,
          int DataItem){
  Circle_Double_List *ptr = Head;
  //допоміжний покажчик
  do {
```

```
    if (DataItem == ptr->Data)
      return true;
    else ptr = ptr->Next;
    }
  while (ptr != Head);
  return false;
}
```

//перевірка порожнини циклічного двоспрямованого списку
```
bool Empty_Circle_Double_List(Circle_Double_List* Head){
  return (Head != NULL ? false : true);
}
```

//виключення циклічного двоспрямованого списку
```
void Delete_Circle_Double_List(Circle_Double_List* Head){
  if (Head != NULL){
    Head = Delete_Item_Circle_Double_List(Head, 1);
    Delete_Circle_Double_List(Head);
  }
}
```

Контрольні питання

1. У чому переваги і недоліки організації структур у вигляді стека?
2. У чому переваги і недоліки організації структур у вигляді черги?
3. Для моделювання яких реальних завдань зручно використовувати стек? А для яких чергу?
4. Яке значення зберігає покажчик на стек?
5. Яке значення зберігає покажчик на чергу?
6. Які існують обмеження на тип інформаційного поля стеки і черги?
7. З якою метою в програмах виконується перевірка на порожнини стека і черги?
8. При роботі зі стеком або чергою доступні позиції обмеженого числа елементів. Чи можлива ситуація запису нових елементів стека або черги на вже зайняті власними елементами ділянки пам'яті (запис поверх себе)? Відповідь обґрунтуйте.
9. З якою метою в програмах виконується видалення стека і черги після закінчення роботи з ними? Як зміниться робота програми, якщо операцію видалення не виконувати?
10. У чому принципова відмінність лінійного односпрямованого (двоспрямованого) і циклічного односпрямованого (двоспрямованого) списків?
11. Як уникнути зациклення при перегляді циклічного списку?

Лекція 4
БІНАРНІ ДЕРЕВА

Дерево – це структура даних, що являє собою сукупність елементів і відносин, що утворюють ієрархічну структуру цих елементів (рис. 4.1). Кожен елемент дерева називається вершиною (вузлом) дерева. Вершини дерева з'єднані спрямованими дугами, які називають гілками дерева. Початковий вузол дерева називають коренем дерева, йому відповідає нульовий рівень. Листями дерева називають вершини, в які входить одна гілка і не виходить жодної гілки.

Кожне дерево має такі властивості:

1) існує вузол, в який не входить ні одна дуга (корінь);

2) у кожну вершину, крім кореня, входить одна дуга.

Дерева особливо часто використовують на практиці при зображенні різних ієрархій. Наприклад, популярні генеалогічні дерева.

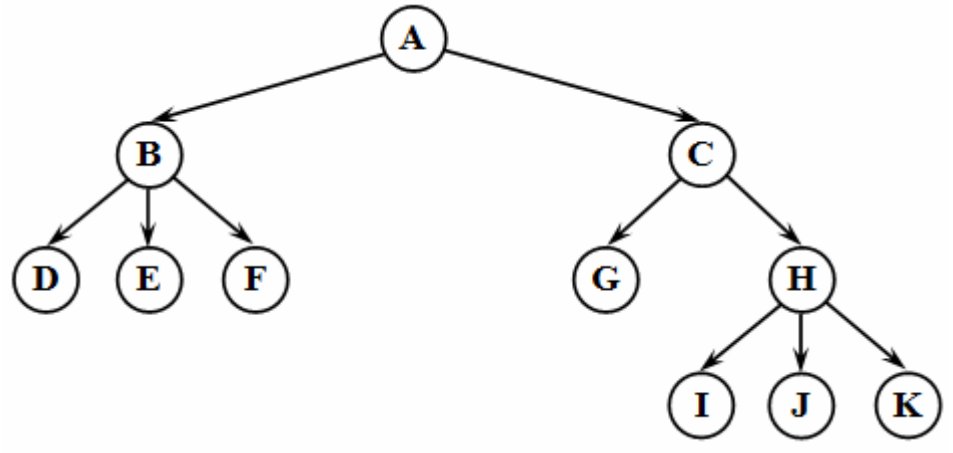


Рисунок 4.1 − Дерево

Всі вершини, до яких входять гілки, що виходять з однієї загальної вершини, називаються нащадками, а сама вершина − предком. Для кожного предка може бути виділено кілька нащадків. Рівень нащадка на одиницю перевищує рівень його предка. Корінь дерева не має предка, а листя дерева не мають нащадків.

Висота (глибина) дерева визначається кількістю рівнів, на яких розташовуються його вершини. Висота порожнього дерева дорівнює нулю, висота дерева з одного кореня − одиниці. На першому рівні дерева може бути тільки одна вершина − корінь дерева, на другому − нащадки кореня дерева, на третьому − нащадки нащадків кореня дерева і т.д.

Піддерево – частина деревовидної структури даних, яка може бути представлена у вигляді окремого дерева.

Ступенем вершини в дереві називається кількість дуг, які з неї виходить. Ступінь дерева дорівнює максимальному ступеню вершини, що входить у дерево. При цьому листями у дереві є вершини, що мають ступінь нуль. За величиною ступеня дерева розрізняють два типи дерев:

− бінарні − ступінь дерева не більше двох;

− сильнорозгалужені − ступінь дерева довільний.

Впорядковане дерево – це дерево, у якого гілки, що виходять з кожної вершини, впорядковані за певним критерієм.

Дерева є рекурсивними структурами, тому що кожне піддерево також є деревом. Таким чином, дерево можна визначити як рекурсивну структуру, в якій кожен елемент є:

– або порожньою структурою;

– або елементом, з яким пов'язано кінцеве число піддерев.

Дії з рекурсивними структурами найзручніше описуються за допомогою рекурсивних алгоритмів.

Уявлення дерев, як списку, засноване на елементах, відповідних вершинам дерева. Кожен елемент має поле даних і два поля покажчиків: покажчик на початок списку нащадків вершини і покажчик на наступний елемент у списку нащадків поточного рівня. За такого способу подання дерева обов'язково слід зберігати покажчик на вершину, що є коренем дерева.

Для того, щоб виконати певну операцію над усіма вершинами дерева необхідно всі його вершини переглянути. Така дія називається обходом дерева.

Обхід дерева − це впорядкована послідовність вершин дерева, в якій кожна вершина зустрічається тільки один раз.

При обході всі вершини дерева повинні відвідуватися у певному порядку. Існує кілька способів обходу всіх вершин дерева. Виділимо три найбільш часто використовуваних способів обходу дерева (рис. 4.2):

– прямий;
– симетричний;
– зворотний

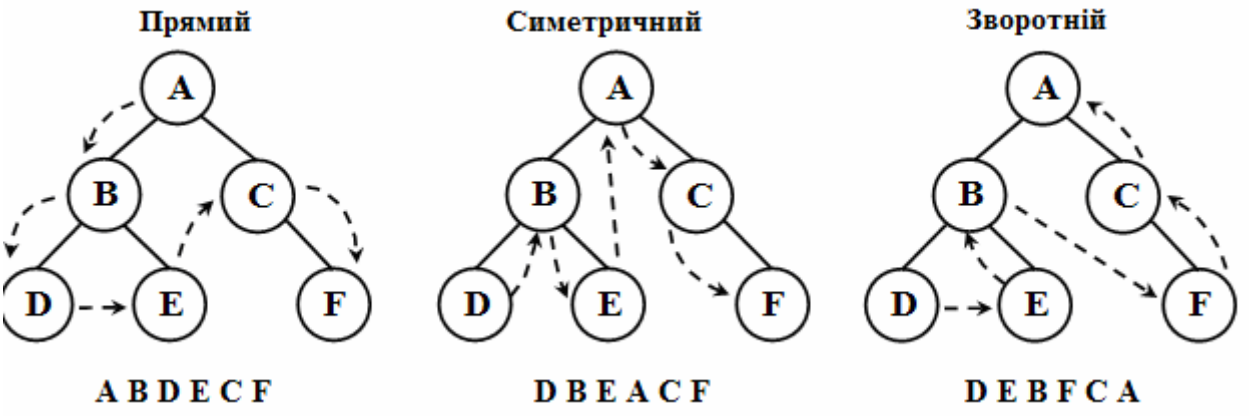


Рисунок 4.2 − Обхід дерев

Існує велике різноманіття деревовидних структур даних. Виділимо найпоширеніші з них: бінарні (двійкові) дерева, червоно-чорні дерева, B-дерева, АВЛ-дерева, матричні дерева, змішані дерева і т.д.

Бінарні дерева

Бінарні дерева є деревами зі ступенем не більше двох.

Бінарне (двійкове) дерево − це динамічна структура даних, що являє собою дерево, в якому кожна вершина має не більше двох нащадків (рис. 4.3).

Таким чином, бінарне дерево складається з елементів, кожен з яких містить інформаційне поле і не більше двох посилань на різні бінарні піддерева. На кожен елемент дерева є рівно одне посилання

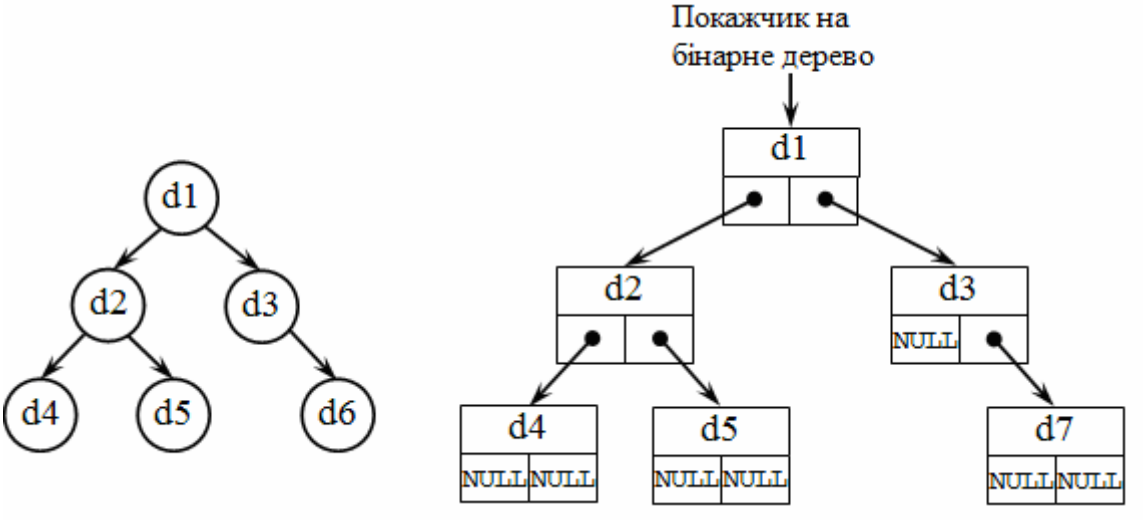


Рисунок 4.3 − Бінарне дерево та його створення

Кожна вершина бінарного дерева є структурою, що складається з чотирьох видів полів. Вмістом цих полів будуть відповідно:

– інформаційне поле (ключ вершини);
– службове поле (їх може бути декілька або жодного);
– покажчик на ліве піддерево;
– покажчик на праве піддерево.

За ступенем вершин бінарні дерева поділяються на (рис. 4.4):

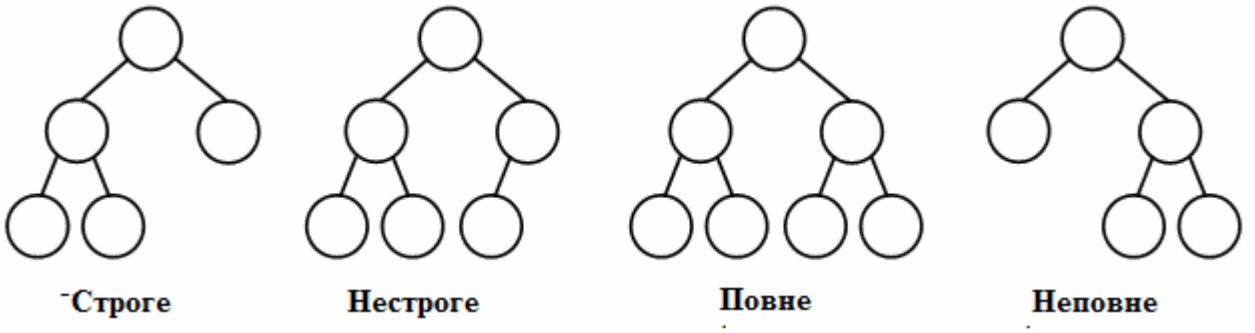


Рисунок 4.4 −Види бінарних дерев

– строге – вершини дерева мають ступінь нуль (у листях) або два (у вузлах);

– нестроге – вершини дерева мають ступінь нуль (у листях), один або два (у вузлах).

У загальному випадку у бінарного дерева на k-му рівні може бути до $2k–1$ вершин. Бінарне дерево називається *повним*, якщо воно містить тільки повністю заповнені рівні. В іншому випадку воно є *неповним.*

Дерево називається *збалансованим*, якщо довжини всіх шляхів від кореня до зовнішніх вершин рівні між собою. Дерево називається *майже*

збалансованим, якщо довжини всіляких шляхів від кореня до зовнішніх вершин відрізняються не більше, ніж на одиницю.

Бінарне дерево може являти собою порожню множину. Бінарне дерево може виродитися у список (рис. 4.5).

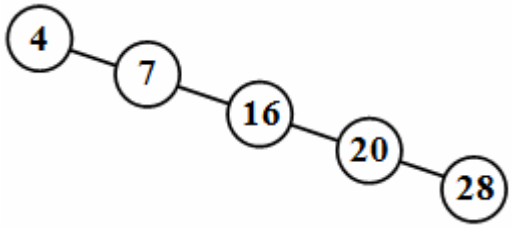


Рисунок 4.5 Список як окремий випадок бінарного дерева

Структура дерева відбивається у вхідному потоці даних таким чином: кожному порожньому зв'язку, який додається, відповідає умовний символ, наприклад, '*' (зірочка). При цьому спочатку описуються ліві нащадки, потім праві. Для структури бінарного дерева, показанного на рис. 4.6, вхідний потік має вигляд: ABD * G *** CE ** FH ** J **.

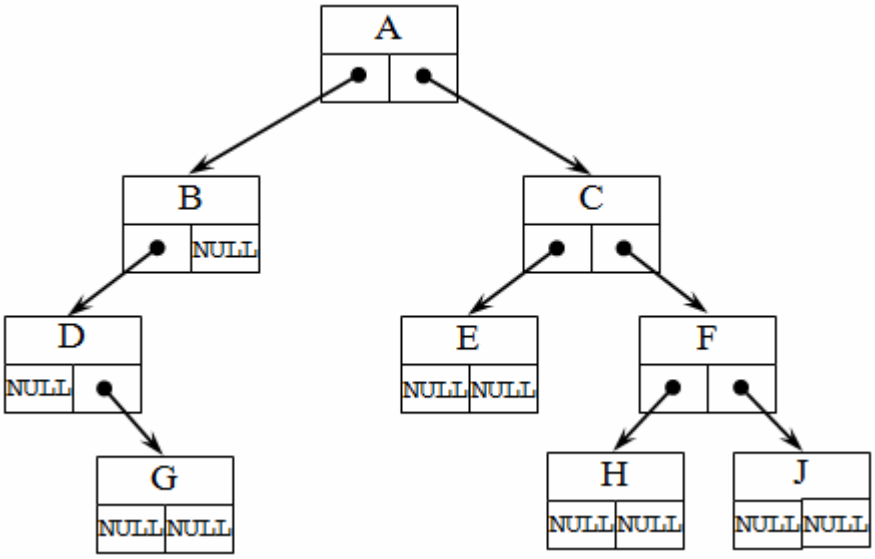


Рисунок 4.6 − Адресація у бінарному дереві

Бінарні дерева можуть застосовуватися для пошуку даних у спеціально побудованих деревах (бази даних), сортування даних, обчислень арифметичних виразів, кодування (метод Хаффмана) і т.д.

Оголошення бінарного дерева виглядає наступним чином:

```
struct ім'я_типу {
        інформаційне поле;
        [Службове поле;]
        адреса лівого піддерева;
        адреса правого піддерева;
       };
```

де інформаційне поле – це поле будь-якого раніше оголошеного або стандартного типу; адреса лівого (правого) піддерева - це покажчик на об'єкт

того самого типу, яким і визначається структура, в нього записується адреса наступного елемента лівого (правого) піддерева.
Наприклад:

```
struct point {
               int data; // інформаційне поле
               int count; // службове поле
               point * left; // адреса лівого піддерева
               point * right; // адреса правого піддерева
             };
```

Основними операціями, які здійснюються з бінарними деревами, є:

- створення бінарного дерева;
- друк бінарного дерева;
- обхід бінарного дерева;
- вставлення елемента в бінарне дерево;
- виключення елемента з бінарного дерева;
- перевірка порожнини бінарного дерева;
- видалення бінарного дерева.

Для опису алгоритмів цих основних операцій використовується наступне оголошення:

```
struct BinaryTree {
   int Data; // поле даних
   BinaryTree * Left; // покажчик на лівий нащадок
   BinaryTree * Right; //покажчик на правий нащадок
};
.........
BinaryTree * BTree = NULL;
```

Дамо функції основних операцій при роботі з бінарним деревом.

```
// створення бінарного дерева
void Make_Binary_Tree(BinaryTree** Node, int n){
  BinaryTree** ptr;//допоміжний покажчик
  srand(time(NULL)*1000);
  while (n > 0) {
    ptr = Node;
    while (*ptr != NULL) {
      if ((double) rand()/RAND_MAX < 0.5)
        ptr = &((*ptr)->Left);
      else ptr = &((*ptr)->Right);
    }
    (*ptr) = new BinaryTree();
    cout << "Введіть значення ";
    cin >> (*ptr)->Data;
    n--;
  }
}
```

//друк бінарного дерева
```
void Print_BinaryTree(BinaryTree* Node, int l){
  int i;
  if (Node != NULL) {
    Print_BinaryTree(Node->Right, l+1);
    for (i=0; i< l; i++) cout << "    ";
    printf ("%4ld", Node->Data);
    Print_BinaryTree(Node->Left, l+1);
  }
  else cout << endl;
}
```

//прямий обхід бінарного дерева
```
void PreOrder_BinaryTree(BinaryTree* Node){
  if (Node != NULL) {
    printf ("%3ld",Node->Data);
    PreOrder_BinaryTree(Node->Left);
    PreOrder_BinaryTree(Node->Right);
  }
}
```

//зворотний обхід бінарного дерева
```
void PostOrder_BinaryTree(BinaryTree* Node){
  if (Node != NULL) {
    PostOrder_BinaryTree(Node->Left);
    PostOrder_BinaryTree(Node->Right);
    printf ("%3ld",Node->Data);
  }
}
```

//симетричний обхід бінарного дерева
```
void SymmetricOrder_BinaryTree(BinaryTree* Node){
  if (Node != NULL) {
    PostOrder_BinaryTree(Node->Left);
    printf ("%3ld",Node->Data);
    PostOrder_BinaryTree(Node->Right);
  }
}
```

//включення вершини до бінарного дерева
```
void Insert_Node_BinaryTree(BinaryTree** Node,int Data) {
  BinaryTree* New_Node = new BinaryTree;
  New_Node->Data = Data;
  New_Node->Left = NULL;
  New_Node->Right = NULL;
  BinaryTree** ptr = Node;//допоміжний покажчик
  srand(time(NULL)*1000);
```

```
  while (*ptr != NULL) {
    double q = (double) rand()/RAND_MAX;
    if ( q < 1/3.0) ptr = &((*ptr)->Left);
    else if ( q > 2/3.0) ptr = &((*ptr)->Right);
    else break;
  }
  if (*ptr != NULL) {
    if ( (double) rand()/RAND_MAX < 0.5 )
      New_Node->Left = *ptr;
    else New_Node->Right = *ptr;
    *ptr = New_Node;
  }
  else{
    *ptr = New_Node;
  }
}
```

//виключення вершини з бінарного дерева

```
void Delete_Node_BinaryTree(BinaryTree** Node,int Data){
  if ( (*Node) != NULL ){
    if ((*Node)->Data == Data){
      BinaryTree* ptr = (*Node);
   if ( (*Node)->Left == NULL && (*Node)->Right == NULL )
(*Node) = NULL;
    else if ((*Node)->Left == NULL) (*Node) = ptr->Right;
    else if ((*Node)->Right == NULL) (*Node) = ptr->Left;
      else {
        (*Node) = ptr->Right;
        BinaryTree ** ptr1;
        ptr1 = Node;
        while (*ptr1 != NULL)
          ptr1 = &((*ptr1)->Left);
        (*ptr1) = ptr->Left;
      }
      delete(ptr);
      Delete_Node_BinaryTree(Node,Data);
    }
    else {
      Delete_Node_BinaryTree(&((*Node)->Left),Data);
      Delete_Node_BinaryTree(&((*Node)->Right),Data);
    }
  }
}
```

//перевірка порожнини бінарного дерева

```
bool Empty_BinaryTree(BinaryTree* Node){
  return ( Node == NULL ? true : false );
```

```
}

//звільнення пам’яті, яку було виділено під бінарне дерево
void Delete_BinaryTree(BinaryTree* Node){
  if (Node != NULL) {
    Delete_BinaryTree(Node->Left);
    Delete_BinaryTree(Node->Right);
    delete(Node);
  }
}
```

Червоно-чорні дерева

Бінарні дерева працюють найкраще, коли вони збалансовані, коли довжина шляху від кореня до будь-якого з листя знаходиться в певних межах, пов'язаних з числом вершин. Червоно-чорні дерева є одним з способів балансування дерев. Назва походить від стандартної розмальовки вузлів таких дерев у червоний і чорний кольори. Кольори вершин використовуються при балансуванні дерева.

Червоно-чорне дерево (Red-Black-Tree, RB-Tree) – це бінарне дерево з наступними властивостями (рис. 4.7):

– кожна вершина повинна бути пофарбована або в чорний, або в червоний колір;

– корінь дерева повинен бути чорним;

– листя дерева повинні бути чорними і оголошуватися як NIL-вершини (NIL-вузли, тобто "віртуальні" вузли, спадкоємці вузлів, які зазвичай називають листями; на них "вказують" NULL покажчики);

– кожен червоний вузол повинен мати чорного предка;

– на всіх гілках дерева, що ведуть від його кореня до листя, число чорних вершин однакове.

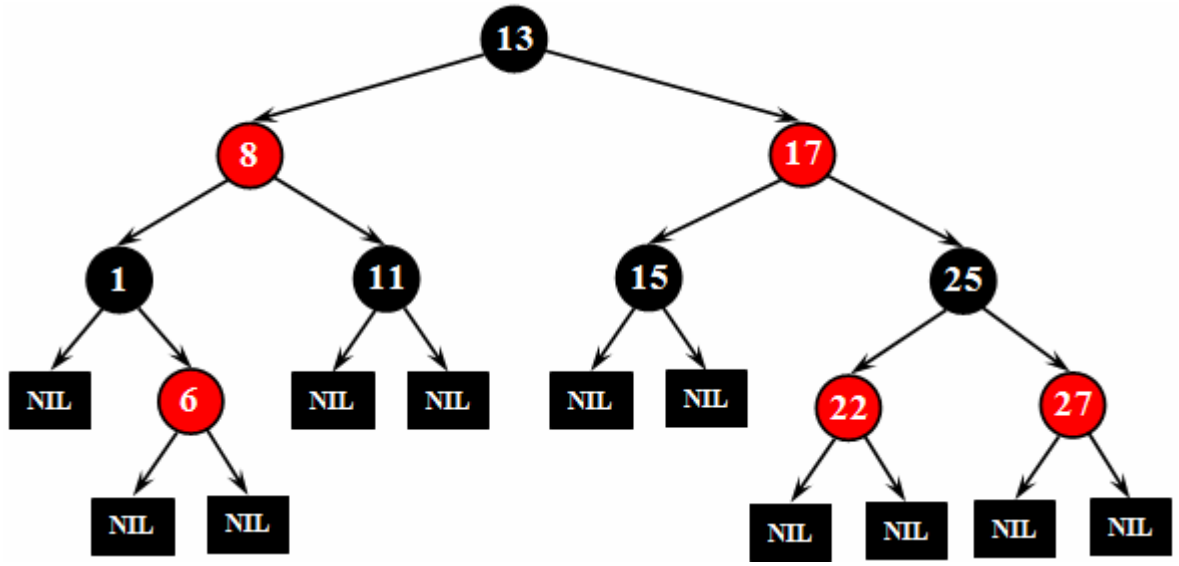


Рисунок 4.7 – Червоно-чорне дерево

Кількість чорних вершин на гілці від кореня до листя називається *чорною висотою дерева*. Перераховані властивості гарантують, що найдовша гілка від кореня до листя не більше ніж удвічі довша будь-якої іншої гілки від кореня до листя.

Над червоно-чорними деревами можна виконувати всі ті ж основні операції, що і над бінарними деревами.

Наведемо функції наступних операцій над червоно-чорними деревами: створення дерева, друк (перегляд) дерева, обхід дерева, перевірка порожнини дерева і видалення дерева.

```
// створення червоно-чорного дерева
void Make_RBTree(RBTree** Node, int n){
  int Data;
  while (n > 0) {
    cout << "Введіть значення ";
        cin >> Data;
    Insert_Node(Node, Data);
    n--;
  }
}

//включення вузла до червого-чорного дерева
void Insert_Node(RBTree** Node,int Data) {
  RBTree **Curent, *Parent, *New_Node;
  Curent = Node;
  Parent = NIL;
  //пошук місця знаходження елемента
  while (*Curent != NIL) {
    Parent = (*Curent);
    Curent = Data < (*Curent)->Data ? &((*Curent)-
>Left) : &((*Curent)->Right);
  }
  // створення нового вузла
  New_Node = new RBTree();
  New_Node->Data = Data;
  New_Node->Parent = Parent;
  New_Node->Left = NIL;
  New_Node->Right = NIL;
  New_Node->color = RED;
  // включення елемента до дерева
  if(Parent != NIL){
    if (Data < Parent->Data) Parent->Left = New_Node;
    else Parent->Right = New_Node;
  }
  else (*Curent) = New_Node;
  Insert_Fixup(Node, New_Node);
```

```
  }

  // підтримка балансу дерева після включення нового елемента
  void Insert_Fixup(RBTree** Node,RBTree* New_Node){
    RBTree* Current = New_Node;
    // перевірка властивостей дерева
    while (Current != *(Node) && Current->Parent->color
== RED){
      //  якщо є порушення
    if (Current->Parent == Current->Parent->Parent-
>Left) {
        RBTree *ptr = Current->Parent->Parent->Right;
        if (ptr->color == RED) {
          Current->Parent->color = BLACK;
          ptr->color = BLACK;
          Current->Parent->Parent->color = RED;
          Current = Current->Parent->Parent;
        }
        else {
          if (Current == Current->Parent->Right) {
            // зробити Current лівим нащадком
            Current = Current->Parent;
            Rotate_Left(Node,Current);
          }
          // перефарбувати та повернути
          Current->Parent->color = BLACK;
          Current->Parent->Parent->color = RED;
          Rotate_Right(Node,Current->Parent->Parent);
        }
      }
      else {
        RBTree *ptr = Current->Parent->Parent->Left;
        if (ptr->color == RED) {
          Current->Parent->color = BLACK;
          ptr->color = BLACK;
          Current->Parent->Parent->color = RED;
          Current = Current->Parent->Parent;
        }
        else {
          if (Current == Current->Parent->Left) {
            Current = Current->Parent;
            Rotate_Right(Node,Current);
          }
          Current->Parent->color = BLACK;
          Current->Parent->Parent->color = RED;
```

```
        Rotate_Left(Node,Current->Parent->Parent);
      }
    }
  }
  (*Node)->color = BLACK;
}
```

//поворот вузла Current вліво

```
void Rotate_Left(RBTree** Node,RBTree *Current) {
  RBTree *ptr = Current->Right;
  Current->Right = ptr->Left;
  if (ptr->Left != NIL) ptr->Left->Parent = Current;
  if (ptr != NIL) ptr->Parent = Current->Parent;
  if (Current->Parent != NIL) {
    if (Current == Current->Parent->Left)
      Current->Parent->Left = ptr;
    else
      Current->Parent->Right = ptr;
    }
  else {
    (*Node) = ptr;
  }
  ptr->Left = Current;
  if (Current != NIL) Current->Parent = ptr;
}
```

//поворот вузла Current вправо

```
void Rotate_Right(RBTree** Node,RBTree *Current) {
  RBTree *ptr = Current->Left;
  Current->Left = ptr->Right;
  if (ptr->Right != NIL) ptr->Right->Parent = Current;
  if (ptr != NIL) ptr->Parent = Current->Parent;
  if (Current->Parent != NIL) {
    if (Current == Current->Parent->Right)
      Current->Parent->Right = ptr;
    else
      Current->Parent->Left = ptr;
  }
  else {
    (*Node) = ptr;
  }
  ptr->Right = Current;
  if (Current != NIL) Current->Parent = ptr;
}
```

```
//друк червоно-чорного дерева
void Print_RBTree(RBTree* Node, int l){
  int i;
  if (Node != NIL) {
    Print_RBTree(Node->Right, l+1);
    for (i=0; i< l; i++) cout << "    ";
    if (Node->color == RED)
      SetConsoleTextAttribute(hStd,FOREGROUND_RED);
    cprintf ("%4ld", Node->Data);
    SetConsoleTextAttribute(hStd,atr);
    Print_RBTree(Node->Left, l+1);
  }
  else cout << endl;
}

//прямий обхід червоно-чорного дерева
void PreOrder_RBTree(RBTree* Node){
  if (Node != NIL) {
    printf ("%3ld",Node->Data);
    PreOrder_RBTree(Node->Left);
    PreOrder_RBTree(Node->Right);
  }
}

//зворотний обхід червоно-чорного дерева
void PostOrder_RBTree(RBTree* Node){
  if (Node != NIL) {
    PostOrder_RBTree(Node->Left);
    PostOrder_RBTree(Node->Right);
    printf ("%3ld",Node->Data);
  }
}

//симетричний обхід червоно-чорного дерева
void SymmetricOrder_RBTree(RBTree* Node){
  if (Node != NIL) {
    PostOrder_RBTree(Node->Left);
    printf ("%3ld",Node->Data);
    PostOrder_RBTree(Node->Right);
  }
}

//перевірка порожнини червоно-чорного дерева
bool Empty_RBTree(RBTree* Node){
  return ( Node == NIL ? true : false );
```

```
}

//звільнення пам’яті, яка виділена під червоно-чорне дерево
void Delete_RBTree(RBTree* Node){
  if (Node != NIL) {
    Delete_RBTree(Node->Left);
    Delete_RBTree(Node->Right);
    delete(Node);
  }
}
```

Контрольні питання

1. З чим пов'язана популярність використання дерев у програмуванні?
2. Чи можна список віднести до дерев? Відповідь обґрунтуйте.
3. Які дані містять адресні поля елемента бінарного дерева?
4. Чи може бінарне дерево бути строгим і неповним? Відповідь обґрунтуйте.
5. Чи може бінарне дерево бути нестрогим і повним? Відповідь обґрунтуйте.
6. Яким може бути майже збалансоване бінарне дерево: повним, неповним, строгим, нестрогим? Відповідь обґрунтуйте.
7. Куди може бути доданий елемент у бінарне дерево у залежності від його виду (повне, неповне, суворе, несурове)? Вид дерева при цьому повинен зберегтися.
8. Куди може бути доданий елемент у збалансоване бінарне дерево? Вид дерева при цьому повинен зберегтися.
9. Чим відрізняються, з точки зору реалізації алгоритму, прямий, симетричний і зворотний обходи бінарного дерева?
10. На підставі чого в червоно-чорному дереві найдовша гілка від кореня до листя не більше ніж удвічі довше будь-який іншої гілки від кореня до листя?
11. Як можна охарактеризувати червоно-чорне дерево: повне, неповне, суворе, несурове?
12. Яким чином при видаленні елемента з червоно-чорного дерева перефарбовуються вузли?

ЛІТЕРАТУРА

1. Альфред Ахо. Структуры данных и алгоритмы/ [Альфред Ахо, Джон Э. Хопкрофт, Джеффри Д.Ульман]. − М.: Вильямс, 2007. − 400 с.
2. Вирт Н. Алгоритмы и структуры данных/ Вирт Н. − М., 2012. − 272 с.
3. Седжвик Н. Фундаментальные алгоритмы на С++. Части 1-4/ Седжвик Н.− Диасофт, 2001. − 688 с.
4. Кнут Д. Искусство программирования. Т. 1-4/ Кнут Д.− М.: Вильямс, 2006. − 682 с.
5. Леонов Ю.Г. Программирование на алгоритмическом языке С++: учебное пособие по лабораторному практикуму/ Леонов Ю.Г., Силкина Н.В., Шпинова Е.Д. – Одесса: ОНАС, 2002. − 68 с.
6. Березин Б.Н. Учебный курс С и С++/ Березин Б.Н., Березин С.Б. – М.: Диалог-МИФИ, 2000. – 288 с.

ЗМІСТ