ЛАБОРАТОРНАТ РОБОТА №12

Бінарні дерева та алгоритми роботи із ними

ТЕОРЕТИЧНІ ВІДОМОСТІ

1. Дерево.

Дерево – в інформатиці та програмуванні одна з найпоширеніших структур даних. Формально дерево визначається як скінченна множина T з однією або більше вершин (вузлів, nodes), яке задовольняє наступним вимогам:

- ✓ існує один відокремлений вузол – корінь (*root*) дерева
- ✓ інші вузли (за виключенням кореня) розподілені серед $m \geq 0$ непересічних множин $T_1, ..., T_m$ і кожна з цих множин в свою чергу є деревом. Дерева $T_1, ..., T_m$ мають назву піддерев (*subtrees*) даного кореня.

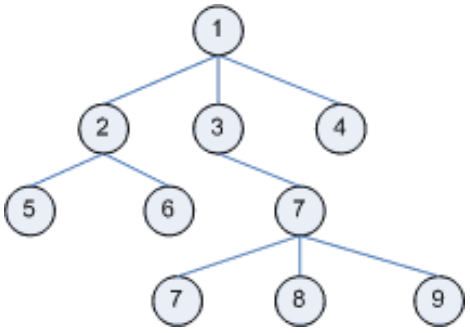

Рис.12.1. Дерево

Найчастіше дерева в інформатиці зображують з коренем, який знаходиться зверху (говорять, що дерево в інформатиці "росте вниз").

2. **Бінарне дерево.**

Важливим окремим випадком кореневих дерев є бінарні дерева, які широко застосовуються в програмуванні і визначаються як множина вершин, яка має відокремлений корінь та два піддерева (праве та ліве), що не перетинаються, або є пустою множиною вершин (на відміну від звичайного дерева, яке не може бути пустим).

Кожна вершина бінарного дерева, відмінна від кореня, може розглядатися як корінь бінарного ***піддерева*** з вершинами, які є досяжними з неї.

Зобразимо декілька бінарних дерев

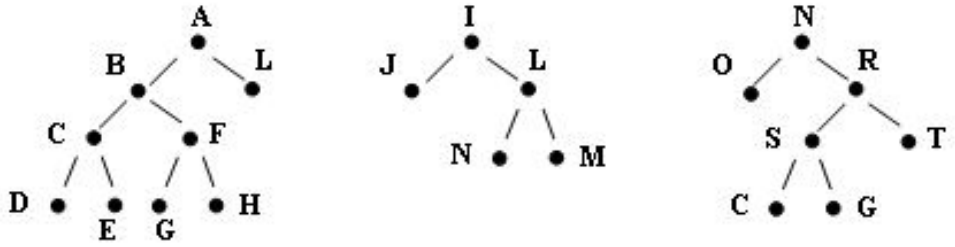

Рис.12.2. Приклади бінарних дерев

В програмуванні бінарне дерево – дерево структура даних, в якому кожна вершина має не більше двох дітей. Зазвичай такі діти називаються правим та лівим. На базі бінарних дерев будуються такі структури, як бінарні дерева пошуку та бінарні купи.

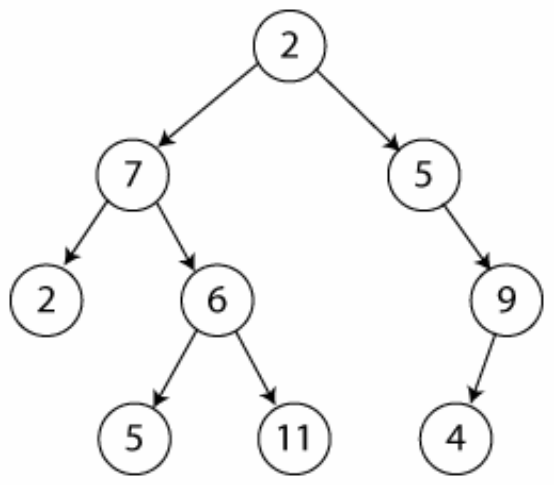

Рис.12.3. Бінарне дерево

✓ Різновиди бінарних дерев

Бінарне дерево – таке кореневе дерево, в якому кожна вершина має не більше двох дітей.

Повне (закінчене) бінарне дерево – таке бінарне дерево, в якому кожна вершина має нуль або двох дітей.

Ідеальне бінарне дерево – це таке повне бінарне дерево, в якому листя (вершини без дітей) лежать на однаковій глибині (відстані від кореня).

Бінарне дерево на кожному n-му рівні має від 1 до $2n$ вершин.

✓ Обхід бінарного дерева

Часто виникає необхідність обійти усі вершини дерева для аналізу інформації, що в них знаходиться. Існують декілька порядків такого обходу, кожний з яких має певні властивості, важливі в тих чи інших алгоритмах: прямий (*preorder*), центрований (*inorder*) та зворотній (*postorder*).

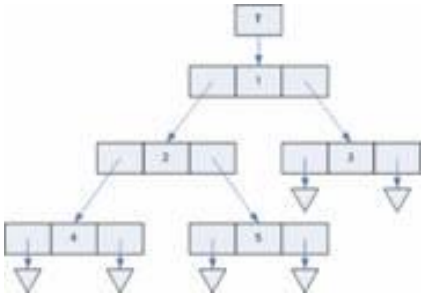

Рис.12.4. Реалізація бінарних дерев

Реалізація бінарного дерева. Кожна вершина містить вказівники на праву та ліву дитину (*left* та *right*)

Існують декілька варіантів конструювання бінарних дерев в залежності від задач, які вирішуються цими структурами та можливостей тої чи іншої мови програмування. Реалізація з використанням вказівників передбачає зберігання в кожній вершині дерева x разом з даними двох полів з вказівниками (правим та лівим) $right[x]$ та $left[x]$ на відповідних дітей цієї вершини.

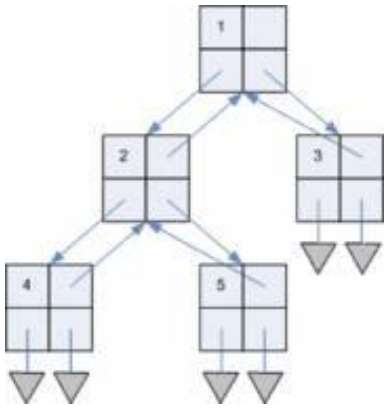

Рис.12.5. Модифікована реалізація бінарного дерева

Модифікована реалізація бінарного дерева. Кожна вершина містить також вказівник на батьківську вершину

Також іноді додається вказівник $p[x]$ на батьківську вершину. Це виявляється корисним, коли необхідний швидкий доступ до батьківської вершини та спрощує деякі алгоритми. Іноді достатньо тільки вказівника на батьківську вершину. Взагалі будь-яке орієнтоване дерево можна описати, знаючи тільки зв’ язки від дітей до батьківської вершини. Деякі різновиди бінарних дерев (наприклад, червоно-чорні дерева або), вимагають збереження в вершинах і деякої додаткової інформації. Якщо у вершини відсутня одна чи обидві дитини, відповідні вказівники ініціалізуються спеціальними "пустими" значеннями.

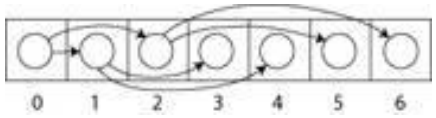

Рис.12.6. *AVL*-дерево

✓ Бінарне дерево на базі масиву

Бінарні дерева також можуть бути побудовані на базі масивів. Такий метод набагато ефективніший щодо економії пам’яті. В такому представленні, якщо вершина має порядковий номер i, то її діти знаходяться за індексами $2i+1$ та $2i+2$, а батьківська вершина за індексом $\frac{i-1}{2}$ (за умов, що коренева вершина має індекс *0*). Інший варіант зберігання дерева в масиві – зберігати індекси дітей.

✓ Представлення n-арних дерев як бінарних

Існує єдине та взаємооднозначне відображення довільного впорядкованого дерева в бінарне.

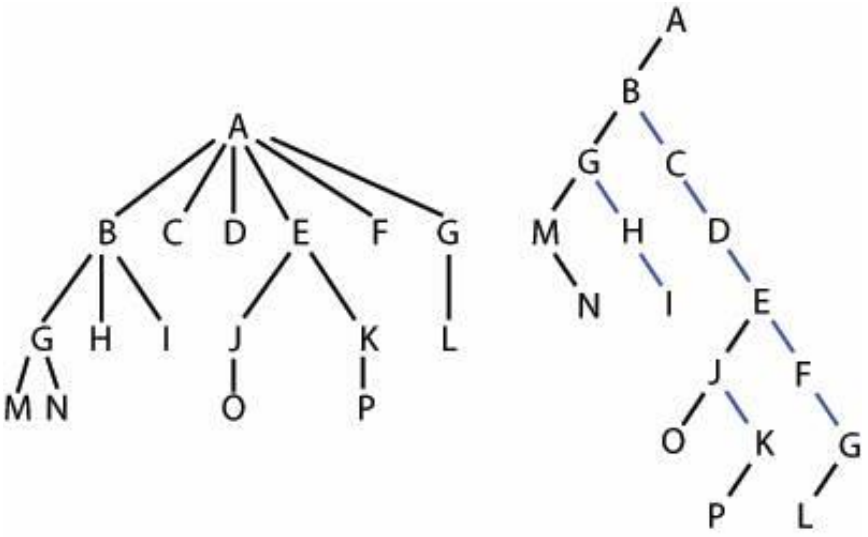

Рис.12.7. Природна відповідність між n-арним деревом та бінарним

Для цього слід послідовно зв’язати усіх дітей кожної сім’ї з першою дитиною та видалити усі вертикальні з’єднання за виключенням з’єднання батька з першою дитиною в сім’ї. Тобто кожна вершина N впорядкованого n-арного дерева відповідає вершині M деякого бінарного дерева. Ліва дитина вершини M відповідає першій дитині вершини N, а права дитина M відповідає першому з наступних братів N (тобто першому з наступних дітей батька вершини N).

Така відповідність має назву природної відповідності між n-арним та бінарним деревом.

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

- інформаційне поле (ключ вершини)
- службове поле (їх може бути декілька!)
- вказівник на ліве піддерево
- вказівник на праве піддерево.

Таким чином, кожна вершина бінарного дерева описуються на мові *C++* таким чином:

```
struct node {
  int Key;            //Ключ вершини.
  int Count;          //Лічильник кількості вершин з однаковими ключами.
  node *Left;         //Вказівник на "лівого" сина.
  node *Right; };     //Вказівник на "правого" сина.
```

3. Побудова бінарного дерева пошуку.

Приведемо спочатку нерекурсивний алгоритм побудови дерева пошуку. В процесі викладу буде роз' яснено, якими властивостями повинне володіти бінарне дерево, щоб бути деревом пошуку.

Позначимо T***ree*** – вказівник на корінь дерева.

```
Tree = NULL; //Побудова порожнього дерева.
```

Позначимо p – допоміжний вказівник на вершину дерева.

Нехай ключ першої, такої вершини, що поступає в дерево, рівний *100*. Створюємо першу вершину

```
p = new(node);
(*p).Key = 100;
(*p).Count = 1;
(*p).Left = NULL;
(*p).Right = NULL;
Tree = p;
```

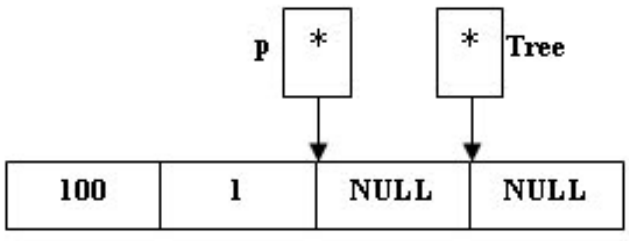

Рис.12.8. Створення першої вершини

Нехай ключ другої, такої вершини, що поступає в дерево, рівний *50*. Виконуємо наступні операції:

- ✓ спочатку створюємо нову вершину

```
p = new(node);
(*p).Key = 50;
(*p).Count = 1;
(*p).Left = NULL;
(*p).Right = NULL;
```

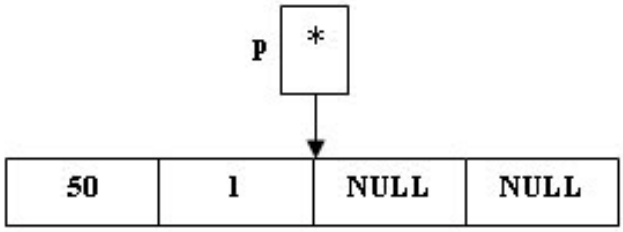

Рис.12.9. Створення нової вершини

- ✓ оскільки *100>50*, то за визначенням бінарного дерева пошуку ми повинні зробити вершину, що знов поступила, лівим сином кореня дерева

```
(*Tree).Left = p;
```

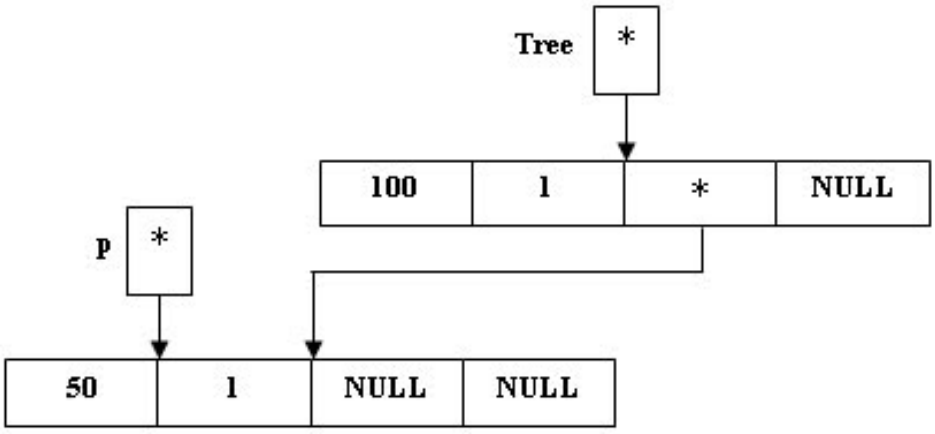

Рис.12.10. Розміщення вершини в лівому піддереві

Нехай ключ третьої вершини, що поступає в дерево, рівний *200*. Порядок наших дій:

- ✓ створюємо нову вершину

```
p = new(node);
(*p).Key = 200;
(*p).Count = 1;
(*p).Left = NULL;
(*p).Right = NULL;
```

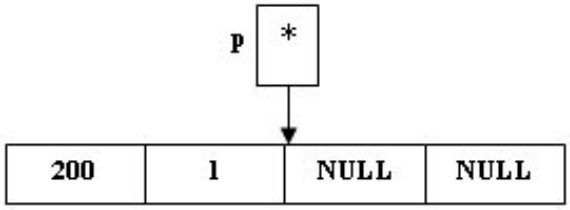

Рис.12.11. Створення нової вершини

✓ оскільки *200>100*, то за визначенням бінарного дерева пошуку, ми повинні зробити вершину, що знов поступила, правим сином кореня дерева

```
(*Tree).Right = p;
```

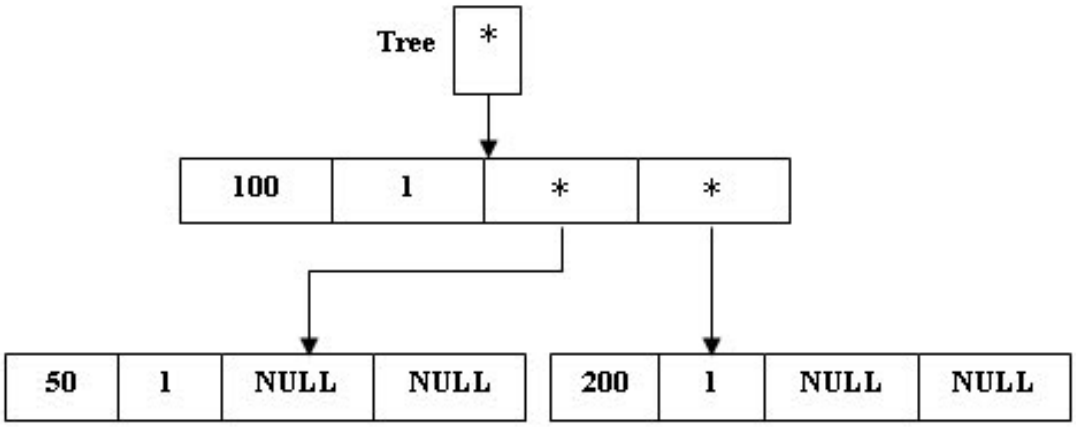

Рис.12.12. Розміщення вершини в правому піддереві

Залишилося визначити, як же нам діяти у разі надходження, наприклад, в дерево знов вершини з ключем *100*. Виявляється, що поле *Count* і застосовується для обліку ключів, що повторюються!

Точніше, в цьому випадку виконується оператор привласнення

```
(*Tree).Count = (*Tree).Count + 1;
```

результат виконання якого зобразимо на схемі

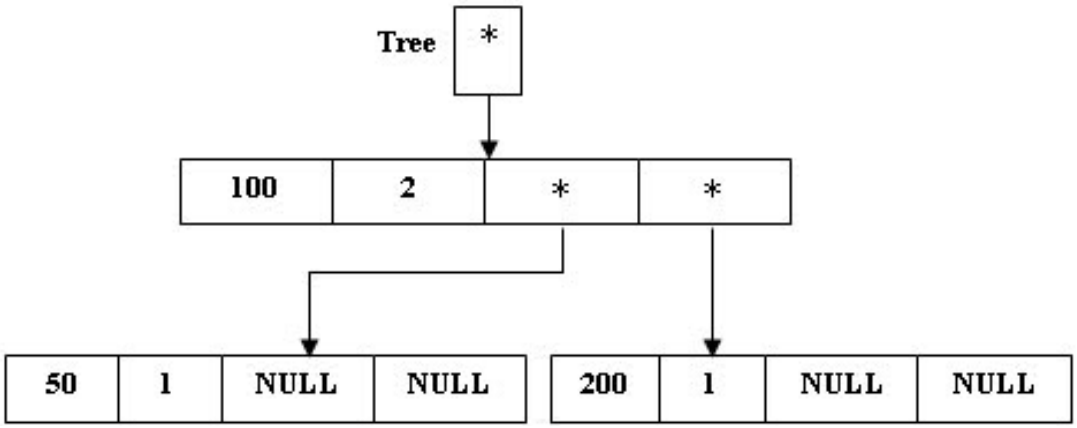

Рис.12.13. Розміщення вершини з ключем, що раніше зустрічався

Таким чином, послідовне надходження вершин з ключами *100, 50, 200, 100* приведе до створення структури даних – бінарного дерева пошуку.

Зрозуміло, що написання алгоритму побудови дерева пошуку з використанням рекурсії або навіть просто розбір рекурсивного алгоритму, написаного кимсь іншим, далеко не просте завдання; вона вимагає великого досвіду. Тому, ми приводимо без коментарів рекурсивну реалізацію алгоритму побудови дерева

```
void BuildTree(node **Tree) {
//Побудова бінарного дерева. *Tree – вказівник на корінь дерева.
  int el;
  *Tree = NULL;       // Побудоване порожнє бінарне дерево.
  cout<<"Введіть ключі вершин дерева...\n";
  cin>>el;
  while(el!=0) { Search (el,Tree); cin>>el;} }
```

У функції BuildTree() використовується функція пошуку вершини з даним ключем *x*

```
void Search(int x, node **p) {
//Пошук вершини з ключем x в дереві зі вставкою (рекурсивний алгоритм).
//*p – вказівник на корінь дерева.
  if(*p==NULL) {
//Вершини з ключем x в дереві немає; включити її.
    *p = new(node); (**p).Key = x; (**p).Count = 1;
    (**p).Left = (**p).Right = NULL; }
  else                              //Поиск місця включення вершини.
    if(x<(**p).Key)                 //Включение в ліве поддерево.
      Search(x,&((**p).Left));
    else if(x>(**p).Key)            //Включение в праве поддерево.
           Search(x,&((**p).Right));
         else (**p).Count = (**p).Count + 1; }
```

Зауваження. Методи пошуку по динамічних таблицях часто називають алгоритмами таблиць символів, оскільки компілятори і інші системні програми зазвичай використовують їх для зберігання визначуваних користувачем символів.

Наприклад, ключем запису в компіляторі може служити символічний ідентифікатор, що позначає деяку змінну в програмі на мовах *Pascal*, *C++* і т.д., а решта полів запису може містити інформацію про тип змінної і її розташування в пам’яті.

Програма пошуку зі вставкою по дереву, яка була приведена на цьому кроці, відмінно підходить для використання як алгоритм таблиць символів, особливо якщо бажано виводити символи в алфавітному порядку.

4. Важливі операції на деревах.

- ✓ обхід вершин в різному порядку;
- ✓ перенумерація вершин;
- ✓ пошук елемента;
- ✓ додавання елемента у визначене місце в дереві;
- ✓ видалення елемента;
- ✓ видалення цілого фрагмента дерева;
- ✓ додавання цілого фрагмента дерева;
- ✓ трансформації (повороти) фрагментів дерева;
- ✓ знаходження кореня для будь-якої вершини;

Найбільшого розповсюдження ці структури даних набули в тих задачах, де необхідне маніпулювання з ієрархічними даними, ефективний пошук в даних, їхнє структуроване зберігання та модифікація.

5. АЛГОРИТМИ ДЛЯ БІНАРНОГО ДЕРЕВА

Для того, щоб проглянути інформаційні поля всіх вершин побудованого дерева, необхідно зробити його обхід (відвідати кожну його вершину).

У разі, коли бінарне дерево порожнє, воно обходиться без виконання будь-яких дій. А для інших випадків існують декілька алгоритмів обходу

5.1. Алгоритм лівобічного обходу дерева:

- ∇ відвідайте корінь дерева;
- ∇ обійдіть ліве піддерево;
- ∇ обійдіть праве піддерево.

Застосовуючи алгоритм лівобічного обходу до бінарних дерев I і II

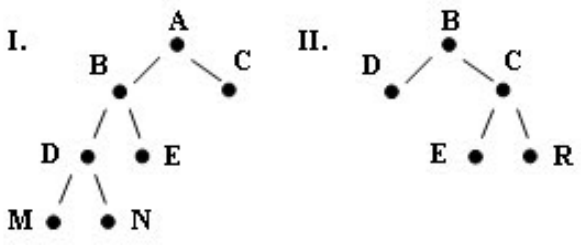

Рис.12.14. Приклади бінарних дерев

відвідаємо вершини в наступному порядку:

I. – *A B D M N E C*

II. – *B D C E R*

Запишемо алгоритм у вигляді рекурсивної функції

```
void ObhodLeft(node **w) {
//Лівобічний обхід дерева. *w – вказівник на корінь дерева.
  if(*w!=NULL) {
    cout<<(**w).Key<<" ";
    ObhodLeft(&((**w).Left));
    ObhodLeft(&((**w).Right)); } }
```

5.2. Алгоритм обходу бінарного дерева від листя до кореня.

Існує ***алгоритм кінцевого обходу дерева***, який полягає в наступному:

- ∇ обійдіть ліве поддерево;
- ∇ обійдіть праве поддерево;
- ∇ відвідайте корінь дерева.

При такому алгоритмі обхід вершин бінарних дерев I, II, зображених на рис.14, відбувається в наступному порядку:

I. – *M N D E B C A*

II. – *D E R C B*

Запишемо алгоритм у вигляді рекурсивної функції

```
void ObhodEnd(node **w) {
//Кінцевий обхід дерева. *w – вказівник на корінь дерева.
  if(*w!=NULL) {
    ObhodEnd(&((**w).Left));
    ObhodEnd(&((**w).Right));
    cout<<(**w).Key<<" "; } }
```

5.3. Алгоритм зворотного обходу бінарного дерева.

І, нарешті, приведемо алгоритм зворотного обходу дерева, який полягає в наступному:

- ∇ обійдіть ліве поддерево;
- ∇ відвідайте корінь дерева;
- ∇ обійдіть праве поддерево.

Застосовуючи цей алгоритм до бінарних дерев I, II, , зображених на рис.14, відбувається в наступному порядку:

I. – *M D N B E A C*

II. – *D B E C R*

Запишемо алгоритм у вигляді рекурсивної функції

```
void ObhodBack(node **w) {
//Зворотний обхід бінарного дерева. *w – вказівник на корінь дерева.
  if(*w!=NULL) {
    ObhodBack(&((**w).Left));
    cout<<(**w).Key<<" ";
    ObhodBack(&((**w).Right)); } }
```

5.4. Алгоритм виведення на екран бінарного дерева.

Для зображення дерева на екрані дисплея використовується обхід дерева, який ми назвемо зворотним зворотному.

Погляньте на приведену нижче функцію

```
void Vyvod(node **w,int l) {
//Зображення дерева w на екрані дисплея (рекурсивний алгоритм).
//*w – вказівник на корінь дерева.
  int i;
  if(*w!=NULL {
    Vyvod(&((**w).Right),l+1);
    for(i=1; i<=l; i++) cout<<" ";
    cout<<(**w).Key<<endl;
    Vyvod(&((**w).Left),l+1); } }
```

Саме дерево "лежить на лівому боці". Спочатку виводиться праве піддерево, причому чергова вершина "відступає" від лівого краю вікна на величину, рівну глибині вершини (відстань від кореня до цієї вершини). Цей відступ реалізується циклом

```
for(i=1; i<=l; i++) cout<<" ";
```

Зверніть увагу, що значення змінної *l* кожного разу збільшується на *1* при рекурсивному зверненні до функції Vyvod().

Далі розглянемо ще декілька нерекурсивних алгоритмів.

5.5. Алгоритм пошуку вершини в бінарному дереві.

```
node Poisk_1(int k, node **Tree) {
//Пошук вершини з ключем до в дереві (рекурсивний алгоритм). *Tree –
//вказівник на вершину дерева. Функція повертає вказівник на вершину,
// що містить ключ до.
  if(*Tree==NULL) return (NULL);
  else
    if((**Tree).Key==k) return (*Tree);
    else {
      if(k<(**Tree).Key) return Poisk_1(к,&((**Tree).Left));
      else return Poisk_1(к,&((**Tree).Right)); } }
```

5.6. Алгоритм додавання вершини в бінарне дерево.

Перед додаванням вершини в дерево звертаємося до функції Poisk(*k,Tree,&r*);. Якщо значення глобальної змінної *B*, яке повернула функція Poisk, рівний *FALSE* (у дереві немає ключа, рівного що знов поступає), то вершина, після якої здійснюватиметься додавання, є листом. Вказівник на цей лист зберігається в змінній *r* .

У *heap*-області резервуємо місце для динамічного об'єкту

```
s = new(node);
(*s).Key = Елем;
(*s).Count = 1;
(*s).Left = (*s).Right = NULL;
```

і "підвішуємо" вершину

```
*Tree = s;.
```

Інакше, якщо вершина, після якої прикріплюватимемо нову, не є листом дерева, то перед "підвішуванням" визначаємо, в яке поддерево щодо цієї вершини буде включена нова

```
if(k<(*r).Key) (*r).Left = s;
else (*r).Right = s;
```

Таким чином, текст функції додавання вершини матиме вигляд

```
void Addition(node **Tree, int k) {
//Додавання вершини до в бінарне дерево. *Tree – вказівник на вершину
//дерева. B – глобальна булева змінна:
                      //TRUE, якщо вершина з ключем до в дереві знайдена
                      //FALSE, інакше.
  node *r, *s;
  Poisk(k,Tree,&r);
  if(!B) {
    s = new(node);
    (*s).Key = k;
    (*s).Count = 1;
    (*s).Left = (*s).Right = NULL;
    if(*Tree==NULL) *Tree = s;
    else
      if(k<(*r).Key) (*r).Left = s;
      else (*r).Right = s; }
  else (*r).Count += 1; }
```

5.7. Алгоритми видалення вершини з бінарного дерева.

Алгоритм видалення з бінарного дерева вершини із заданим ключем розрізняє три випадки:

- ✓ вершини із заданим ключем в дереві немає;

- ✓ вершина із заданим ключем має не більше однієї витікаючої дуги;
- ✓ вершина із заданим ключем має дві витікаючі дуги.

Н.Вірт [3-5] відзначає: "Трудність полягає у видаленні елементів з двома нащадками, оскільки ми не можемо вказати одним посиланням на два напрями. Елемент, що в цьому випадку видаляється, потрібно замінити або на найправіший елемент його лівого поддерева, або на найлівіший елемент його правого поддерева. Ясно, що такі елементи не можуть мати більше одного нащадка. Докладне це показано в рекурсивній процедурі, званій Delete()..."

```
void Delete(node **Tree, int k) {
//Видалення вершини до з бінарного дерева. *Tree – вказівник на вершину дерева.
  node *q;
  if(*Tree==NULL) cout<<"Вершина із заданим ключем не знайдена!\n";
  else
    if(k<(**Tree).Key) Delete(&((**Tree).Left),k);
    else
      if(k>(**Tree).Key) Delete(&((**Tree).Right),k);
      else {
        q = *Tree;
        if((*q).Right==NULL) { *Tree = (*q).Left; delete q; }
        else
          if((*q).Left==NULL) { *Tree = (*q).Right; delete q;}
          else Delete_1(&((*q).Left),&q); } }
```

"Допоміжна рекурсивна процедура Delete_1() викликається тільки в 3-му випадку. Вона "спускається" уздовж найправішої гілки лівого поддерева вузла q, що видаляється, і потім замінює істотну інформацію (ключ і лічильник) в q відповідними значеннями найправішої компоненти r цього лівого поддерева, після чого від r можна звільнитися..." (Н.Вірт). Звільнення пам’ яті, зайнятої динамічної змінної, відбувається за допомогою конструкції delete.

```
void Delete_1(node **r, node **q) {
  node *s;
  if((**r).Right==NULL) {
    (**q).Key = (**r).Key;
    (**q).Count = (**r).Count;
    *q = *r;
    s = *r;
    *r = (**r).Left;
    delete s; }
  else Delete_1(&((**r).Right),q); }
```

6. ПРИКЛАДИ АЛГОРИТМІВ

Приклад 1. Наведемо приклад програми, де реалізується побудова бінарного дерева пошуку, різноманітні його обходи, виведення дерева і визначення його висоти, а також "очищення" дерева (рекурсивні алгоритми).

```
#include<iostream.h>
  struct  node {
    int Key;
    int Count;
    node *Left;
    node *Right; };
//=========================================================================
  class TREE {
  private:
    node *Tree;       //Указатель на корінь дерева.
    void Search(int,node**);
  public:
    TREE() {Tree=NULL;}
    node** GetTree() {return &Tree;} //Получение вершини дерева.
    void BuildTree();
    void CleanTree(node **);
    void ObhodEnd(node **);
    void ObhodLeft(node **);
    void ObhodBack(node **);
    void Vyvod(node**,int);
    int Height(node**); };
//=========================================================================
  void main() {
    TREE A;
    A.BuildTree();
    cout<<"\nВивід дерева:\n";
    A.Vyvod(A.GetTree(),0);
    cout<<"\nВысота дерева:"<<A.Height(A.GetTree())<<endl;
    cout<<"\nЛевосторонний обхід дерева: "; A.ObhodLeft(A.GetTree());
    cout<<"\nКонцевой обхід дерева: "; A.ObhodEnd(A.GetTree());
    cout<<"\nОбратный обхід дерева: ";
    A.ObhodBack(A.GetTree()); A.CleanTree(A.GetTree()); }
//=========================================================================
```

//1.1. Алгоритм побудови бінарного дерева

```
//=========================================================================
  void TREE::BuildTree() {
//Побудова бінарного дерева (рекурсивний алгоритм).
//Tree – вказівник на корінь дерева.
    int el;
    cout<<"Вводите ключі вершин дерева ...\n";
    cin>>el;
    while(el!=0) {
      Search(el,&Tree); cin>>el; } }
```

```
//=========================================================================
```

//1.2. Алгоритм пошуку вершини

```
//=========================================================================
  void TREE::Search(int x,node **p) {
//Пошук вершини з ключем x в дереві зі вставкою (рекурсивний алгоритм).
//*p – вказівник на корінь дерева.
    if(*p==NULL) {
//Вершини в дереві немає; включити її.
      *p = new(node);
      (**p).Key = x; (**p).Count = 1;
      (**p).Left = NULL; (**p).Right = NULL; }
    else
      if(x<(**p).Key) Search(x,&((**p).Left));
      else
        if(x>(**p).Key) Search(x,&((**p).Right));
        else(**p).Count = (**p).Count + 1; }
//=========================================================================
```

//1.3. Алгоритм лівобічного обходу дерева

```
//=========================================================================
  void TREE::ObhodLeft(node **w) {
//Лівобічний обхід дерева. *w – вказівник на корінь дерева.
    if(*w!=NULL) {
      cout<<(**w).Key<<" ";
      ObhodLeft(&((**w).Left));
      ObhodLeft(&((**w).Right)); } }
//=========================================================================
```

//1.4. Алгоритм обходу бінарного дерева від листя до коренння

```
//=========================================================================
  void TREE::ObhodEnd(node **w) {
//Кінцевий обхід дерева. *w – вказівник на корінь дерева.
    if(*w!=NULL) {
      ObhodEnd(&((**w).Left));
      ObhodEnd(&((**w).Right));
      cout<<(**w).Key<<" "; } }
//=========================================================================
```

//1.5. Алгоритм зворотного обходу бінарного дерева

```
//=========================================================================
  void TREE::ObhodBack(node **w) {
//Зворотний обхід дерева. *w – вказівник на корінь дерева.
    if(*w!=NULL) {
      ObhodBack(&((**w).Left));
      cout<<(**w).Key<<" ";
      ObhodBack(&((**w).Right)); } }
//=========================================================================
```

//1.6. Алгоритм очистки бінарного дерева

```
//=========================================================================
  void TREE::CleanTree(node **w) {
//Очищення дерева. *w – вказівник на корінь дерева.
    if(*w!=NULL) {
      CleanTree(&((**w).Left));
      CleanTree(&((**w).Right));
      delete *w; } }
//=========================================================================
```

//1.7. Алгоритм виведення на екран бінарного дерева

```
//=========================================================================
  void TREE::Vyvod(node **w,int l) {
//Зображення дерева *w на екрані дисплея (рекурсивний алгоритм).
//*w – вказівник на корінь дерева.
    int i;
    if(*w!=NULL) {
      Vyvod(&((**w).Right),l+1);
      for(i=1; i<=l; i++) cout<<"   ";
      cout<<(**w).Key<<endl;
      Vyvod(&((**w).Left),l+1); } }
//=========================================================================
```

//1.8. Алгоритм визначення висоти (глибини) бінарного дерева

```
//=========================================================================
  int TREE::Height(node **w) {
//Визначення висоти бінарного дерева. *w – вказівник на корінь дерева.
    int h1,h2;
    if(*w==NULL) return (-1);
    else {
      h1 = Height(&((**w).Left));
      h2 = Height(&((**w).Right));
      if(h1>h2) return (1 + h1);
      else  return (1 + h2); } }
```

Результат роботи програми зображений на рис.15:

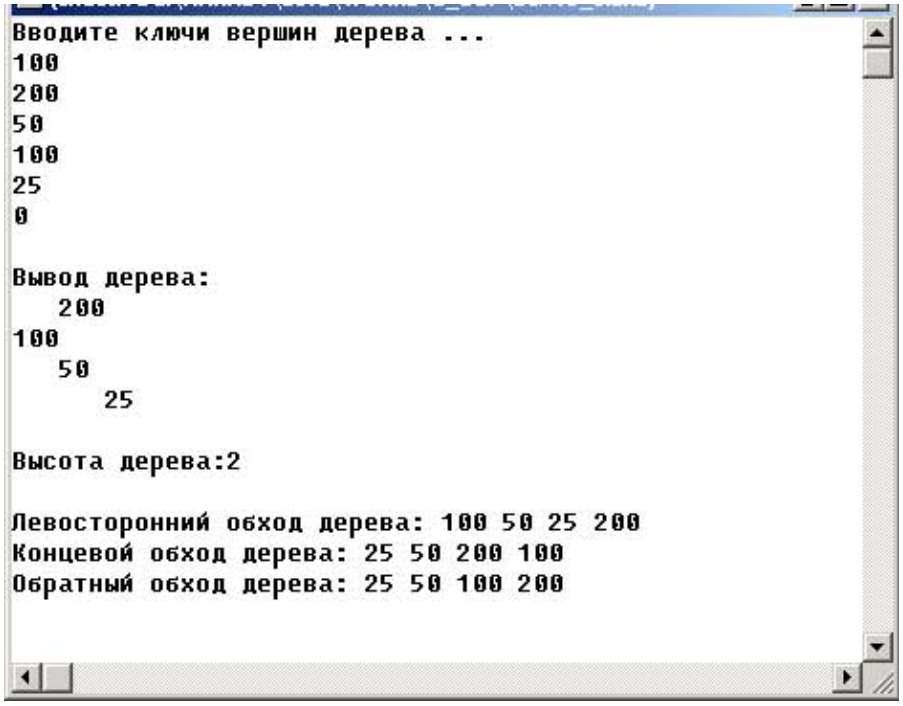

Рис.12.15. Результат роботи додатку

Приклад 2. Наведемо приклад програми, що ілюструє пошук заданої вершини в дереві, додавання вершини в дерево, видалення вершини з дерева.

```
#include<iostream.h>
#define TRUE 1
#define FALSE 0
  struct node {
    int Key;
    int Count;
    node *Left;
    node *Right; };
//=========================================================================
  class TREE {
  private:
    node *Tree;       //Указатель на корінь дерева.
    node  *Res;       //Указатель на знайдену вершину.
    int B;            //Признак знаходження вершини в дереві.
                      //Пошук вершини в дереві (рекурсивний алгоритм).
    void Search(int node**);
    void Delete_1(node**,node**);
  public:
    TREE() { Tree = NULL;}
    node** GetTree() {return &Tree;}
    void BuildTree(); //Построение бінарного дерева.
                      //Виведення дерева на екран (рекурсивний алгоритм).
    void Vyvod(node**,int);
    int Poisk(int);   //Пошук вершини в дереві (нерекурсивний алгоритм).
                      //Пошук вершини в дереві (рекурсивний алгоритм).
    node *Poisk_1(int,node **);
                      //Додавання вершини в дерево (нерекурсивний алгоритм).
    void Addition(int);
```

```
                        //Видалення вершини з дерева.
    void Delete(node**, int); };
//=========================================================================
  void main() {
    TREE A;
    int el;
    A.BuildTree(); A.Vyvod(A.GetTree(),0);
    cout<<"Введите ключ вершини, яку потрібно знайти в дереві: ";
    cin>>el;
    if(A.Poisk (el)) cout<<"В дереві є така вершина!\n";
    else  cout<<"В дереві немає такої вершини!\n";
    cout<<"Введите ключ вершини, яку потрібно знайти в дереві: ";
    cin>>el;
    if(A.Poisk_1(el,A.GetTree())!=NULL)
      cout<<"В дереві є така вершина!\n";
    else  cout<<"В дереві немає такої вершини!\n";
    cout<<"Введите ключ вершини, що додається: ";
    cin>>el;
    A.Addition(el);  A.Vyvod(A.GetTree(),0);
    cout<<"Введите ключ вершини, що видаляється: ";
    cin>>el;
    A.Delete(A.GetTree(),el);  A.Vyvod(A.GetTree(),0); }
//=========================================================================
```

//2.1. Алгоритм побудови бінарного дерева

```
//=========================================================================
  void TREE::BuildTree(){
//Побудова бінарного дерева. Tree – вказівник на вершину дерева.
    int el;
    cout<<"Вводите ключі вершин дерева: \n";
    cin>>el;
    while(el!=0) {
      Search(el,&Tree); cin>>el; } }
//=========================================================================
```

//2.2. Алгоритм виведення на екран бінарного дерева

```
//=========================================================================
  void TREE::Vyvod(node **w,int l) {
//Зображення дерева w на екрані дисплея (рекурсивний алгоритм).
//*w – вказівник на корінь дерева.
    int i;
    if(*w!=NULL) {
      Vyvod(&((**w).Right),l+1);
      for(i=1; i<=l; i++) cout<<"   ";
      cout<<(**w).Key<<endl;
      Vyvod(&((**w).Left),l+1); } }
//=========================================================================
```

//2.3. Алгоритм пошуку ланки

```
//=========================================================================
  void TREE::Search(int x,node **p) {
```

```
//Пошук ланки x в бінарному дереві зі вставкою (рекурсивний алгоритм).
//*p – вказівник на вершину дерева.
    if(*p==NULL) { //Вершини в дереві немає; включити її.
      *p = new(node);
      (**p).Key = x;     (**p).Count = 1;
      (**p).Left = (**p).Right = NULL; }
    else
      if(x<(**p).Key) Search(x,&((**p).Left));
      else
        if(x>(**p).Key) Search(x,&((**p).Right));
        else (**p).Count += 1; }
//=========================================================================
```

//2.4. Алгоритм додавання вершини в бінарне дерево

```
//=========================================================================
  void TREE::Addition(int k) {
//Пошук ланки до в бінарному дереві зі вставкою (нерекурсивний алгоритм).
//Tree – вказівник на вершину дерева.
    node *s;
    Poisk(k);
    if(!B) {
      s = new(node);
      (*s).Key  = k;  (*s).Count = 1;
      (*s).Left = (*s).Right = NULL;
      if(Tree==NULL) Tree = s;
      else
        if(k<(*Res).Key) (*Res).Left = s;
        else  (*Res).Right = s; }
    else  (*Res).Count += 1; }
//=========================================================================
```

//2.5. Перший алгоритм пошуку вершини

```
//=========================================================================
  int TREE::Poisk(int k) {
//Пошук вершини з ключем k в дереві (нерекурсивний алгоритм).
//Tree – вказівник на бінарне дерево.
//Res – вказівник на знайдену вершину або на лист, до якого можна
//приєднати нову вершину.
    node *p,*q;
    B = FALSE; p = Tree;
    if(Tree!=NULL)
      do {
        q = p;
        if((*p).Key==k) B = TRUE;
        else {
          q = p;
          if(k<(*p).Key) p = (*p).Left;
          else  p = (*p).Right; } }
      while(!B && p!=NULL);
    Res = q;
```

```
    return B; }
//=========================================================================
```

//2.6. Другий алгоритм пошуку вершини

```
//=========================================================================
  node *TREE::Poisk_1(int k,node **p) {
//Пошук вершини з ключем k в дереві (рекурсивний алгоритм).
//*p – вказівник на корінь дерева.
    if(*p==NULL) return (NULL);
    else if((**p).Key==k) return (*p);
         else
           if(k<(**p).Key) return Poisk_1(к,&((**p).Left));
           else return Poisk_1(к,&((**p).Right)); }
//=========================================================================
```

//2.7. Перший алгоритм видалення вершини з бінарного дерева

```
//=========================================================================
  void TREE::Delete(node **p,int k) {
//Видалення вершини k з бінарного дерева. *p – вказівник на корінь дерева.
    node *q;
    if(*p==NULL) cout<<"Вершина із заданим ключем не знайдена!\n";
    else
      if(k<(**p).Key) Delete(&((**p).Left),k);
      else
        if(k>(**p).Key) Delete(&((**p).Right),k);
        else {
          q = *p;
          if((*q).Right==NULL) {*p = (*q).Left; delete q;}
          else
            if((*q).Left==NULL) { *p = (*q).Right; delete q; }
            else  Delete_1(&((*q).Left),&q); } }
//=========================================================================
```

//2.8. Другий алгоритм видалення вершини з бінарного дерева

```
//=========================================================================
  void TREE::Delete_1(node **r,node **q) {
    node *s;
    if((**r).Right==NULL) {
      (**q).Key = (**r).Key; (**q).Count = (**r).Count;
      *q = *r; s = *r; *r = (**r).Left; delete s; }
    else  Delete_1(&((**r).Right), q); }
```

ЗАВДАННЯ 12

№	Задача
1	Створіть програмою числове двійкове дерево. Опишіть логічну функцію, перевіряючу наявність заданого числа в сформованому дереві.
2	Створіть програмою числове двійкове дерево. Опишіть числову функцію, що підраховує суму елементів дерева.
3	Створіть програмою числове двійкове дерево. Опишіть функцію, яка знаходить найбільший елемент непорожнього дерева

4	Напишіть програму, процедуру, яка кожний негативний елемент дерева замінює на позитивний, а позитивний перетворює на нуль.
5	Напишіть програму, процедуру, яка кожний елемент дерева підносить до квадрату.
6	Створіть програмою символьне двійкове дерево. Опишіть логічну функцію, перевіряючу наявність заданого числа в сформованому дереві.
7	Створіть двійкове дерево записів. Перевірте вибране поле записів на рівність вказаному елементу.
8	Створіть програмою символьне двійкове дерево. Опишіть логічну функцію, перевіряючу, чи є в непорожньому дереві хоча б два однакові символи.
9	Створіть програмою два числові двійкові дерева. В кожному з них знайдіть максимальні елементи.
10	Створіть рядкове двійкове дерево. Опишіть функцію, повертаючу рядок, сформований на базі символів, що зустрічаються в кожному рядку дерева.
11	Створіть програмою числове двійкове дерево. Опишіть функцію, яка знаходить добуток елементів непорожнього дерева.
12	Створіть програмою символьне двійкове дерево. Створіть аналогічне дерево, яке містить коди символів першого дерева.
13	Створіть програмою числове двійкове дерево. Опишіть функцію, яка знаходить суму елементів непорожнього дерева.
14	Створіть програмою два числові двійкові дерева. В кожному з них знайдіть мінімальні елементи.
15	Напишіть функцію, яка знаходить крайній лівий листок бінарного дерева.
16	Напишіть функцію, яка знаходить крайній правий листок бінарного дерева.
17	Створіть програмою числове двійкове дерево. Опишіть функцію, яка знаходить суму парних елементів непорожнього дерева.
18	Напишіть процедуру, яка рахує кількість листків бінарного дерева.
19	Напишіть програму, процедуру, яка кожний негативний елемент дерева замінює на його квадрат, а позитивний замінює на його корінь квадратний.
20	Створіть програмою числове двійкове дерево. Опишіть функцію, яка знаходить суму непарних елементів непорожнього дерева.