ЛАБОРАТОРНАТ РОБОТА №13

Дерева відрізків та алгоритми роботи із ними

ТЕОРЕТИЧНІ ВІДОМОСТІ

1. Дерево відрізків.

Дерево відрізків – це структура даних, створена для роботи з такими інтервалами на числовій осі, кінці яких належать фіксованій множині з N абсцис. Оскільки множина абсцис фіксована, то дерево відрізків є статичною структурою по відношенню до цих абсцис, тобто структурою, на якій не дозволені вставки і видалення абсцис; крім того ці абсциси можна нормалізувати, замінюючи кожну з них її порядковим номером при обході їх зліва направо. Не втрачаючи сукупності, можна вважати ці абсциси цілими числами в інтервалі $[1,N]$.

Дерево відрізків – це двійкове дерево з коренем. Для заданих цілих чисел l і r таких, що $l<r$**,** дерево відрізків $T(l,r)$ будується рекурсивно таким чином: воно складається з кореня ***v*** з параметрами $B[v]=l$ і $E[v]=r$ (B і E мнемонічно відповідають словам "Beginning" (початок) і "End" (кінець), а якщо $r-l>1$, то воно складається з лівого піддерева $T\left(l,\left(B[v]+E[v]\right)DIV\,2\right)$ і правого піддерева $T\left(\left(\left(B[v]+E[v]\right)DIV\,2\right),r\right)$. Параметри $B[v]$ і $E[v]$ позначають інтервал $\left[B[v],E[v]\right]$, включений в $[l,r]$**,** пов’ язаний з вузлом v.

Приведемо приклад дерева відрізків:

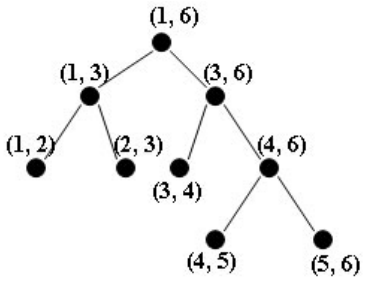

Рис.13.1. Дерево відрізків

Інтервали, що належать множині $\left\{\left[B[v],E[v]\right]: v-вузол\ T(l,r)\right\}$ називаються ***стандартними інтервалами*** дерева $T(l,r)$.

Стандартні інтервали, що належать листку $T(l,r)$, називаються ***елементарними інтервалами***. Строго кажучи, інтервал, пов' язаний з v, це напіввідкритий інтервал $[B[v],E[v]]$, за винятком вузлів найправішого шляху в $T(l,r)$**,** чиї інтервали замкнуті.

Можна безпосередньо переконатися, що $T(l,r)$ ідеально збалансовано (все його листя належить двом суміжним рівням) і має глибину $[\log_2(r-l)]$.

2. ПРИКЛАДИ АЛГОРИТМІВ

*Приклад 1***.** Побудуємо дерева відрізків і виведемо їх на екран дисплея

```
#include <iostream.h>
  struct node {
    int KeyMin;       //Мінімальний ключ вершини.
    int KeyMax;       //Максимальний ключ вершини.
    node *Left;       //Вказівник на "лівого" сина.
    node *Right;      //Вказівник на "правого" сина.
  };
//=========================================================================
  class TREE {
  private:
    node *Tree;       //Вказівник на корінь дерева.
    void Search(int,int,node**);
  public:
    TREE() {Tree = NULL;}
    void BuildTree(); //Побудова дерева відрізків.
    node** GetTree() {
      return &Tree;}  //Отримання вершини дерева.
    void CleanTree(node **);
    void Vyvod(node **,int); };
//=========================================================================
  void main() {
    TREE A;
    A.BuildTree ();
    cout<<"\nВивід дерева:\n";
    A.Vyvod(A.GetTree(),0);
    A.CleanTree(A.GetTree()); }
//=========================================================================
```

//1.1. Алгоритм побудови бінарного дерева

```
//=========================================================================
  void TREE::BuildTree() {
//Побудова бінарного дерева (рекурсивний алгоритм). Tree - вказівник на
//корінь дерева.
    int k1,k2;
    cout<<"Введіть два цілі числа...\n";
```

```
    cin>>k1;
    cin>>k2;
    Search(k1,k2,&Tree); }
//=========================================================================
```

//1.2. Алгоритм побудови відрізків

```
//=========================================================================
  void TREE::Search(int k1, int k2, node **p) {
//Побудова дерева відрізків p. p – вказівник на корінь дерева.
  if (k2-k1>1) {
    *p = new(node);
    (**p).KeyMin = k1;
    (**p).KeyMax = k2;
    (**p).Left = (**p).Right = NULL;
    Search(k1,(k1+k2) /2,&((**p).Left));
    Search((k1+k2)/2,k2,&((**p).Right)); }
  else {
    *p = new(node);
    (**p).KeyMin = k1;
    (**p).KeyMax = k2;
    (**p).Left = (**p).Right = NULL; } }
//=========================================================================
```

//1.3. Алгоритм очищення дерева

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

//1.4. Алгоритм виведення на екран дерева відрізків

```
//=========================================================================
  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).KeyMin<<", "<<(**w).KeyMax<<endl;
    Vyvod(&((**w).Left),l+1); } }
```

Результат роботи програми зображений на рис.2

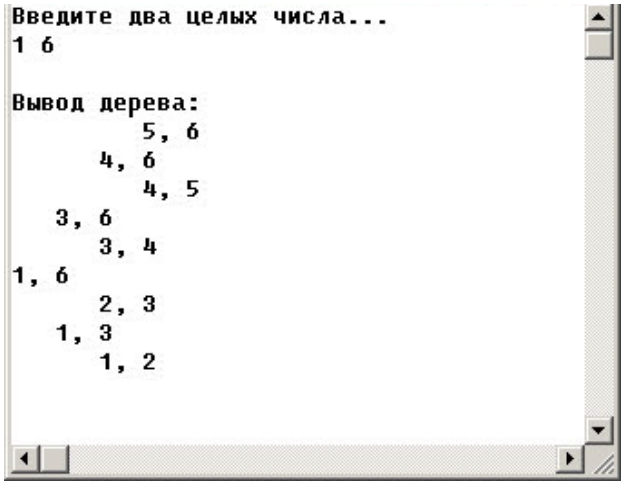

Рис.13.2. Результат роботи додатку

Дерево відрізків $T(l,r)$ призначене для динамічного запам' ятовування тих інтервалів, чиї кінці належать множині $\{l,l+1,...,r\}$. Зокрема, при $r-l>3$ довільний інтервал $[b,e]$ з цілими $b<e$ буде розбитий в набір з не більше ніж $\lceil \log_2(r-l) \rceil + \lceil \log_2(r-l) \rceil - 2$ стандартних інтервалів дерева $T(l,r)$.

Дерево відрізків – надзвичайно гнучка структура даних у зв' язку з численними додатками. Відзначимо тільки, що якщо треба знати число інтервалів, що містять дану точку *X*, то простий двійковий пошук в T (тобто проходження шляху від кореня до листа) повністю вирішує цю задачу.

Приклад 2.

```
#include <iostream.h>
  struct node {
    int KeyMin;       //Мінімальний ключ вершини.
    int KeyMax;       //Максимальний ключ вершини.
    node *Left;       //Вказівник на "лівого" сина.
    node *Right; };   //Вказівник на "правого" сина.
//=========================================================================
  class TREE {
  private:
    node *Tree;       //Вказівник на корінь дерева.
    int S;            //Кількість входжень заданої точки в дерево.
    void Search(int,int,node**);
  public:
    TREE() {Tree = NULL; S = 0;}
    void BuildTree();       //Побудова дерева відрізків.
    node** GetTree() {return &Tree;} //Отримання вершини дерева.
    void CleanTree(node **);
    void Vyvod(node **,int);
    int GetCount() { return S;}
    void Count(node **,float); };
//=========================================================================
```

```
  void main() {
    TREE A;
    float X;
    A.BuildTree();
    cout<<"\nВивід дерева:\n";
    A.Vyvod(A.GetTree(),0);
    cout<<"\nВведіть абсцису точки: ";
    cin>>X;
    A.Count(A.GetTree(),X);
    cout<<"Точка належить "<< A.GetCount()<<" інтервалам";
    A.CleanTree(A.GetTree()); }
//=============================================================================
```

//2.1. Алгоритм побудови бінарного дерева

```
//=============================================================================
  void TREE::BuildTree(){
//Побудова бінарного дерева (рекурсивний алгоритм). Tree – вказівник на
//корінь дерева.
    int k1,k2;
    cout<<"Введіть два цілі числа...\n";
    cin>>k1;
    cin>>k2;
    Search(k1,k2,&Tree); }
//=============================================================================
```

//2.2. Алгоритм побудови відрізків

```
//=============================================================================
  void TREE::Search(int k1, int k2, node **p) {
//Побудова дерева відрізків p. p – вказівник на корінь дерева.
    if(k2-k1>1) {
      *p = new(node);
      (**p).KeyMin = k1;
      (**p).KeyMax = k2;
      (**p).Left = (**p).Right = NULL;
      Search(k1,(k1+k2)/2,&((**p).Left));
      Search((k1+k2)/2,k2,&((**p).Right)); }
    else {
      *p = new(node);
      (**p).KeyMin = k1;
      (**p).KeyMax = k2;
      (**p).Left = (**p).Right = NULL; } }
//=============================================================================
```

//2.3. Алгоритм підрахунку кількості інтервалів

```
//=============================================================================
  void TREE::Count(node **p, float X) {
//Підрахунок кількості інтервалів дерева p, що містять точку X.
    if(*p!=NULL) {
      Count(&((**p).Right),X);
      if(X>=(**p).KeyMin && X<=(**p).KeyMax)  S++;
      Count(&((**p).Left),X); } }
//=============================================================================
```

//2.4. Алгоритм очистки дерева відрізків

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

//2.5. Алгоритм виведення на екран дерева вдрізків

```
//=========================================================================
  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).KeyMin<<", "<<(**w).KeyMax<<endl;
      Vyvod(&((**w).Left),l+1); } }
```

ЗАВДАННЯ 13

Непарні (по списку в журналі) варіанти:

1. Протестувати програму наведену в прикладі 1, пояснити викладачу як вона працює. Відповісти на питання викладача.

Парні (по списку в журналі) варіанти:

2. Протестувати програму наведену в прикладі 2, пояснити викладачу як вона працює. Відповісти на питання викладача.