

# SỬ DỤNG CẤU TRÚC CÂY NHỊ PHÂN- BINARY TREES

## Nội dung

- Một số định nghĩa và tính chất liên quan đến cây nhị phân cần nhớ
- Mô tả một phần tử trên cây và mô tả một cây nhị phân
- Khi nào dùng cây nhị phân
- Tại sao lại là cây nhị phân
- Tóm tắt về phép duyệt cây nhị phân
- Minh hoạ cây không thứ tự
- Minh hoạ cây BST-Binary Search Tree

## 1- Một số định nghĩa và tính chất liên quan đến cây nhị phân cần nhớ

**Cây nhị phân:** Tập nút trong đó có 1 nút gốc, một nút có tối đa 2 nút con.

**Các loại nút:** Nút gốc (root), nút cha (father), nút con (child), nút trên (ancestor), nút dưới (descendant), nút lá (leaf, terminal), nút trung gian/ nút trong (internal)

**Đường đi- path:** Đường duy nhất từ nút gốc đến nút đang được xét.

**Mức (level)** của một nút, **chiều cao (height)** của cây.

**Cây có thứ tự- ordered tree:** cây chứa tập trị thoả mãn một tiêu chuẩn định trước.

**Cây BST:** Cây nhị phân có thứ tự với điều kiện: trị nút này sẽ lớn trị nút con trái và nhỏ hơn trị của nút con phải. Chính nội dung các tác vụ thêm/xoá/ sửa của cây giúp xác định cây này cây thông dụng/ cây có thứ tự hay không (BST là một cây có thứ tự).

**Một số quan hệ trên cây nhị phân.**

- Số cạnh = số nút – 1
- Số nút là  $\geq$  số
- Số nút tại mức L  $n(L) \leq 2^L$
- Số nút lá của cây có chiều cao H:  $\text{leaves} \leq 2^H$
- Số nút của một cây nhị phân đầy đủ với i nút trung gian:  $n = 2i+1$
- Số nút lá c ủa một cây nhị phân với i nút trung gian:  $\text{leaves} = i+1$

## 2- Mô tả một phần tử trên cây và mô tả một cây nhị phân

```
class BinTreeNode <T> {  
    T info;  
    Node<T> left;  
    Node<T> right;  
    .....  
}
```

```
class BinTree <T> {  
    BinTreeNode<T> root ;  
    .....  
    .....  
    .....  
}
```

**Nhân xét:** Info trong nút nên có các đặc điểm:

- Có khả năng so sánh để có thể mở rộng thành nút được dùng cho các tình huống cần sắp xếp ( nút trong cây BST chẳng hạn)
- Có trị nào đó nhằm phân biệt duy nhất (key) để giúp quá trình tìm kiếm với điều kiện là có tối đa một kết quả.

## 3- Khi nào dùng cây nhị phân

- Dữ liệu có phân cấp tự nhiên như: danh sách gia phả, tập trạng thái trong các trò chơi.
- Dữ liệu có phân cấp dạng cấu trúc: Dữ liệu được trình bày trên cửa sổ trình duyệt ( cây DOM, Document Object Model)....

#### 4- Tại sao lại là cây nhị phân

Dĩ nhiên chúng ta có thể dùng cây n-phân. Nếu dùng cây n-phân, tại mỗi nút chúng ta phải lưu trữ một mảng  $n$  tham khảo đến  $n$  nút con nhưng không phải lúc nào cũng có đủ  $n$  con  $\rightarrow$  phí bộ nhớ.

Chúng ta hoàn toàn có thể dùng cây nhị phân để biểu diễn cây  $n$  phân với ý nghĩa của các tham chiếu như sau:

- **Tham chiếu left** chỉ đến nút con đầu
  - **Tham chiếu right** chỉ đến nút anh em cùng cha.
- $\Rightarrow$  Hai tham chiếu này mang ý nghĩa khác với hai tham chiếu được đề cập trong cây nhị phân ban đầu  $\rightarrow$  Các thuật toán trên cây phải được viết lại cho phù hợp.

#### 5- Tóm tắt về phép duyệt cây- Traversing

**Duyệt cây:** Quá trình viếng thăm từng nút trong một cây.

##### **Cơ chế duyệt cây:**

Cả cây nhị phân chỉ được quản lý bằng nút gốc, mọi con đường đều đi từ nút gốc và theo nguyên tắc biết nút cha mới biết được nút con nên mọi cách duyệt đều có nguyên tắc chung là:

- Tại một thời điểm chỉ viếng thăm một nút.
- Trật tự các nút sẽ được viếng thăm phải có cách đi đến bằng cách lưu trữ nút sẽ đi đến vào stack hệ thống( cơ chế hàm đệ quy) hoặc stack/ hoặc queue tự quản lý.

##### - **Các cơ chế thống dụng**

**Duyệt theo mức**  $\rightarrow$  Duyệt theo chiều rộng/ chiều ngang của cây ( Breadth-first traversal): Khi duyệt 1 nút thì cất các nút con vào queue.

**Duyệt theo nhánh**  $\rightarrow$  Duyệt theo độ sâu, Depth-first traversal  $\rightarrow$  Dùng kỹ thuật đệ quy và có 6 thứ tự duyệt (V: visit, viếng thăm nút hiện hành) VLR, VRL, LVR, RVL, LRV, RLV – Dùng stack hệ thống (hàm đệ quy) hoặc kỹ thuật lập có sự hỗ trợ của stack tự tạo.

**Biến thể:** Duyệt có xâu kết cả một nhánh ( Threads Tree), Morris traversal

**Chính tác vụ duyệt cây là cơ bản nhất để giải hầu hết các tác vụ trên cây dù cây đó là cây có hoặc không có thứ tự**

( chi tiết các phép duyệt này đã được đề cập chi tiết trong sách giáo khoa)

**Cần chú ý rằng các phép duyệt cây là chung cho mọi cây kể cả cây BSY. BST là một cây đặc biệt nhằm giải quyết tốt các bài toán quản lý dữ liệu dạng cây có yêu cầu về tìm kiếm sao cho hiệu quả ( $O(\log n)$ ) nên các tác vụ chèn/ xóa/ tìm kiếm được hiệu chỉnh phù hợp với thứ tự dữ liệu đã được ấn định trước.**

#### 6- Minh họa về cây không thứ tự

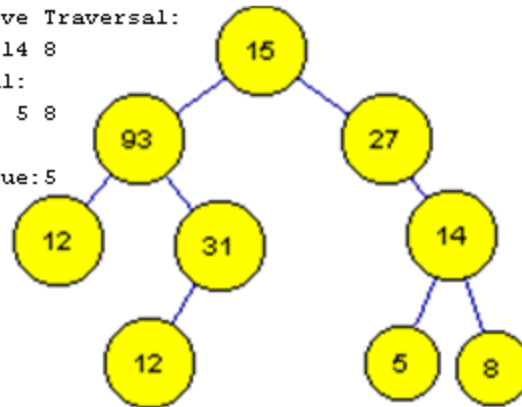
##### Thí dụ

Thí dụ sau minh họa cách dùng pháp duyệt cây để nhập vào 1 cây số nguyên sau đó thực thi các thao tác như xuất cây (theo bề sâu và bề rộng), tìm chiều cao của cây, tìm kiếm trên cây, in ra các nút trong một mức. Kết quả của một lần chạy như hình sau:

```

Input a tree:
Enter an integer for a node, 0 for quit: 15
Left of 15: Enter an integer for a node, 0 for quit: 93
Left of 93: Enter an integer for a node, 0 for quit: 12
Left of 12: Enter an integer for a node, 0 for quit: 0
Right of 12: Enter an integer for a node, 0 for quit: 0
Right of 93: Enter an integer for a node, 0 for quit: 31
Left of 31: Enter an integer for a node, 0 for quit: 12
Left of 12: Enter an integer for a node, 0 for quit: 0
Right of 12: Enter an integer for a node, 0 for quit: 0
Right of 31: Enter an integer for a node, 0 for quit: 0
Right of 15: Enter an integer for a node, 0 for quit: 27
Left of 27: Enter an integer for a node, 0 for quit: 0
Right of 27: Enter an integer for a node, 0 for quit: 14
Left of 14: Enter an integer for a node, 0 for quit: 5
Left of 5: Enter an integer for a node, 0 for quit: 0
Right of 5: Enter an integer for a node, 0 for quit: 0
Right of 14: Enter an integer for a node, 0 for quit: 8
Left of 8: Enter an integer for a node, 0 for quit: 0
Right of 8: Enter an integer for a node, 0 for quit: 0
Inorder-LNR- Recursive Traversal:
12 93 12 31 15 27 5 14 8
Bread-first Traversal:
15 93 27 12 31 14 12 5 8
Height: 3
Input a searched value: 5
The value 5 exists.
Input a level: 3
12 5 8

```



```

MyQueue.java x
1  /* java.util.Queue là interface
2     * ==> Xây dựng lớp cho Queue, dùng LinkedList đã có
3     * Lớp này được dùng trong phép duyệt cây BREADTH-FIRST
4     */
5  package trees;
6  import java.util.LinkedList;
7  public class MyQueue<E> extends LinkedList<E> {
8      public MyQueue() { super();}
9      // thêm vào cuối hàng đợi
10     public void enqueue ( E x) { this.addLast(x); }
11     public E dequeue() { return this.poll(); }
12
13 }

```

```
IntBinTreeNode.java x
1  /* Lớp mô tả cho một nút trong cây nhị phân các số nguyên */
2  package trees;
3  public class IntBinTreeNode {
4      int key; // khoá giúp phân biệt dữ liệu trong nút
5      IntBinTreeNode left, right;
6  public IntBinTreeNode() { left = right = null; }
7  public IntBinTreeNode(int k) { key=k; left = right = null; }
8      // tạo 1 nút với 2 con đã biết
9  public IntBinTreeNode(int k, IntBinTreeNode left, IntBinTreeNode right) {
10     key=k; this.left = left; this.right= right; }
11     // Getters, setters
12     public int getKey() {...}
13     public void setKey(int key) {...}
14     public IntBinTreeNode getLeft() {...}
15     public void setLeft(IntBinTreeNode left) {...}
16     public IntBinTreeNode getRight() {...}
17     public void setRight(IntBinTreeNode right) {...}
18 }
30
```

```
IntBinTree.java x
1  /* Lớp mô tả cây nhị phân số nguyên */
2  package trees;
3  import java.util.Scanner;
4  public class IntBinTree {
5      final int LEFT = 0;
6      final int RIGHT = 1;
7      IntBinTreeNode root;
8      public static Scanner sc= new Scanner(System.in);
9  public IntBinTree() { root=null; }
10     // Viếng 1 nút p trên cây
11     protected void visit( IntBinTreeNode p){
12         System.out.print(p.key + " ");
13     }
14     // CHÚ Ý: Dùng kỹ thuật đệ quy thường phải viết 2 methods
15     // Duyệt inorder nút p - Left Node Right
16     public void inorder( IntBinTreeNode p){
17         if (p!=null){
18             inorder(p.left); // Duyệt nút con trái -LEFT
19             visit(p);
20             inorder(p.right); // Duyệt nút con phải -RIGHT
21         }
22     }
23     // Duyệt cây LNR
24     public void inorder( ){
25         inorder(root);
26     }
27 }
```

```

27 // Duyệt tìm 1 key trên cây con gốc p inorder nút p - NLR
28 public IntBinTreeNode search_inorder( int key, IntBinTreeNode p){
29     IntBinTreeNode result= p;
30     if (p==null) return null;
31     else if (p.key==key) return p;
32     else {
33         result= search_inorder(key, p.left);
34         if (result==null) result=search_inorder(key, p.right);
35     }
36     return result;
37 }
38 // Duyệt cây LNR
39 public IntBinTreeNode search_inorder(int key ){
40     return search_inorder(key, root);
41 }
42 // các methods khác bạn tự xây dựng
43 //

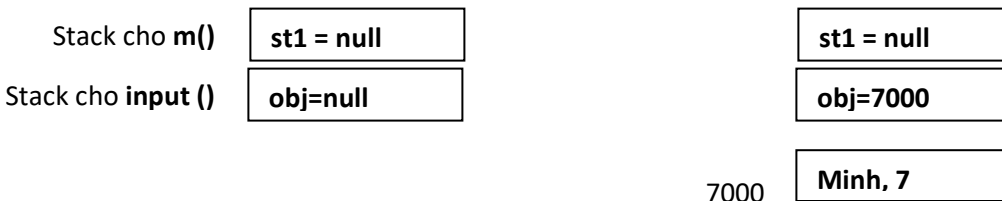
```

### Chú ý khi nhập dữ liệu vào cây

Ngôn ngữ C++ hỗ trợ tham số dạng tham chiếu nên chúng ta có cơ hội làm biến đổi đối số ngoài đã truyền cho hàm. Do vậy, chúng ta có thể truyền **đối số null** từ ngoài hàm để rồi địa chỉ được cấp phát động bên trong hàm sẽ cập nhật vào đối số null bên ngoài hàm. Trong khi đó, Java chỉ hỗ trợ truyền tham số cho hàm dạng tham trị nên khi truyền tham số cho hàm là một tham khảo mang trị **null** thì trị **null** này được chép vào tham số. Việc cấp phát động bên trong hàm không truyền lại giá trị địa chỉ cho đối số ngoài → Đối số bên ngoài vẫn là **null**. Hình sau minh họa việc không thay đổi đối số dạng tham khảo trong Java.

Xét phương thức <b>input(...)</b> trong Java	Và dùng hàm này trong hành vi <b>m()</b> như sau:
<pre> void input( Student obj) {     obj= new Student ("Minh", 7);     .... } </pre>	<pre> void m() {     Student st1= null; //1     input (st1);        //2     ... } </pre>

Bộ nhớ stack khi phát biểu phương thức **m()** thực thi như sau:



**Trước phát biểu `obj= new Student(...)`**

**Sau phát biểu `obj= new Student(...)`**

Như vậy, sau khi hành vi **input(...)** thực thi xong, tham khảo **st1** vẫn mang trị **null**.

**Rút kinh nghiệm:** Hai cách để nhập

Cách 1: Cấp phát bộ nhớ trước khi nhập trị vào các đối tượng. Nếu việc nhập trị này gây ra dữ liệu không được chấp nhận thì hủy đối tượng này đi sau khi nhập.

Cách 2: Hàm nhập trả về trị tham khảo đến đối tượng mới được cấp phát để có thể gán lại cho đối số ngoài( tham khảo hai hành vi **input(...)** dưới đây.

```

44 // Nhập cây số nguyên từ bàn phím
45 protected IntBinTreeNode input(IntBinTreeNode p){
46     int x; // trị nhập
47     System.out.print("Enter an integer for a node, 0 for quit: ");
48     x= Integer.parseInt(sc.nextLine());
49     if (x!=0 && p==null) { // chỉ đưa vào trị khác 0
50         p= new IntBinTreeNode(x); // NODE
51         System.out.print("Left of " + x + ":"); // nhập cây con trái
52         p.left= input(p.left); // LEFT
53         System.out.print("Right of " + x + ":"); // nhập cây con phải
54         p.right=input(p.right); // RIGHT
55     }
56     return p;
57 }
58 public void input(){
59     root=null; // huỷ cây cũ nếu có để nhập cây mới
60     root= input(root);
61 }
62 // Breadth-first traversal - Duyệt theo hàng ngang
63 public void breadthFirst(){
64     if (root==null) {
65         System.out.println("Empty tree!");
66         return;
67     }
68     IntBinTreeNode p=root;
69     MyQueue queue = new MyQueue();
70     queue.enqueue(p); // khởi tạo queue là nút gốc để bắt đầu duyệt
71     while (!queue.isEmpty()) { // khi chưa duyệt xong
72         p= (IntBinTreeNode) queue.dequeue(); // lấy 1 nút ra khỏi hàng
73         visit(p); // viếng nút này
74         // cất 2 nút con vào queue
75         if ( p.left!=null) queue.enqueue(p.left);
76         if ( p.right!=null) queue.enqueue(p.right);
77     }
78 }
79 // Duyệt nút p ở mức L với mức biết trước cần duyệt level
80 protected void inorder_level(IntBinTreeNode p, int L, int level){
81     if (p!=null) {
82         if (L==level) visit(p);
83         else if (L<level){
84             inorder_level(p.left, L+1, level);
85             inorder_level(p.right, L+1, level);
86         }
87     }
88 }
89 public void inorder_level(int level) {
90     inorder_level( root, 0, level); // nút gốc có mức 0
91 }
92 // Duyệt tìm chiều cao của cây
93 protected int height(IntBinTreeNode p){
94     if ((p==null) || (p.left== null && p.right==null)) return 0;
95     int hL = 1+ height(p.left);
96     int hR = 1+ height(p.right);
97     return hL > hR? hL: hR;
98 }

```

```

99      // tìm chiều cao của cây
100     public int height() {
101         return height(root);
102     }
103
104     public static void main (String[] args){
105         IntBinTree tree = new IntBinTree();
106         System.out.println("Input a tree:");
107         tree.input();
108         System.out.println("Inorder-LMR- Recursive Traversal:");
109         tree.inorder();
110         System.out.println();
111         System.out.println("Bread-first Traversal:");
112         tree.breadthFirst();
113         System.out.println();
114         int h=tree.height();
115         System.out.println("Height: " + h);
116         int x; // trị cần tìm
117         System.out.print("Input a searched value:");
118         x= Integer.parseInt(sc.nextLine());
119         IntBinTreeNode p= tree.search_inorder(x);
120         if (p==null) System.out.println("The value " + x + " does not exist!");
121         else System.out.println("The value " + x + " exists.");
122         int level;
123         System.out.println("Input a level:");
124         level= Integer.parseInt(sc.nextLine());
125         if (level>h) System.out.println("This level does not exist!");
126         else tree.inorder_level(level);
127     }
128 }

```

## 7- Minh họa về cây BST các số nguyên

Chương trình sau minh họa cách sử dụng một cây BST các số nguyên. Ban đầu, khởi tạo một cây BST cân bằng mang các trị 1..10. Sau đó cho phép user thêm/xóa phần tử, xem cây.

### Kết quả một lần chạy chương trình

```

1-Thêm phần tử
2-Xóa phần tử - Delete by Merging
3-Xuất cây dạng giống hạng
4-Xuất cây - breadth-first
5-Xuất cây - Morris Inoder
6-Thoát

```

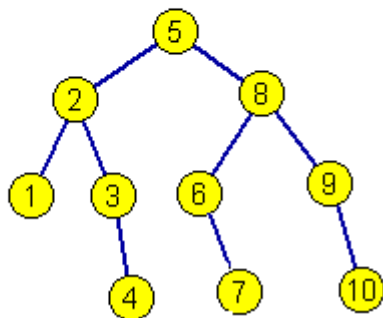
Select an option : 1..6:3

5

```

1__2
 |__1
 |__3
 |__4
1__8
 |__6
 |__7
 |__9
 |__10

```



```

1-Thêm phần tử
2-Xóa phần tử - Delete by Merging
3-Xuất cây dạng giống hạng
4-Xuất cây - breadth-first
5-Xuất cây - Morris Inoder
6-Thoát

```

Select an option : 1..6:4

5 2 8 1 3 6 9 4 7 10

```

1-Thêm phần tử
2-Xóa phần tử - Delete by Merging
3-Xuất cây dạng giống hạng
4-Xuất cây - breadth-first
5-Xuất cây - Morris Inoder
6-Thoát

```

Select an option : 1..6:5

1 2 3 4 5 6 7 8 9 10

Select an option : 1..6:1  
Nhập trị cần thêm vào cây:19

Select an option : 1..6:2  
Nhập trị cần xoá khỏi cây:9

.....  
Select an option : 1..6:3

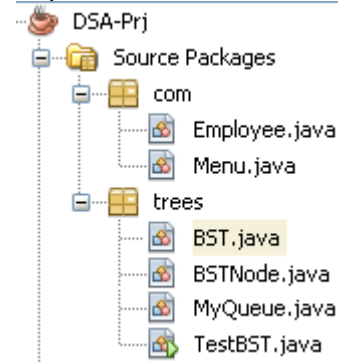
```
5
|__2
|   |__1
|   |   |__3
|   |   |   |__4
|__8
|   |__6
|   |   |__7
|   |   |   |__9
|   |   |   |   |__10
|   |   |   |   |   |__19
```

.....  
Select an option : 1..6:3

```
5
|__2
|   |__1
|   |   |__3
|   |   |   |__4
|__8
|   |__6
|   |   |__7
|   |   |   |__10
|   |   |   |   |__19
```

## Các lớp cần xây dựng

### Lớp Menu



**Lớp Menu:** Đã xây dựng từ bài trước

**Lớp BSTNode:** Mô tả chung cho 1 nút trên cây BST. Vì data trong nút có thứ tự nên buộc lớp con sau này phải implements interface Comparable.

**Lớp BST:** Mô tả cho 1 cây BST có dữ liệu BSTNode

**Lớp MyQueue** mô tả 1 hàng đợi đã có ở demo trước.

**Lớp TestBST:** Cây có dữ liệu Integer. Lớp Integer đã implements interface Comparable rồi.

```
1  /* Lớp cho một nút TỔNG QUÁT trong cây BST
2     * Vì cây có thứ tự nên dữ liệu trên cây phải có khả năng so sánh
3     * => nút thực tế phải hiện thực interface: java.lang.Comparable
4         <T extends Comparable>
5     và có lớp chặn dưới là lớp T này    <T extends Comparable<? super T>>
6
7     */
8     package trees;
9     public class BSTNode<T extends Comparable<? super T>> {
10         protected T el;
11         protected BSTNode<T> left, right;
12         public BSTNode() { left = right = null; }
13         public BSTNode(T el) { this(el,null,null); }
14         public BSTNode(T el, BSTNode<T> lt, BSTNode<T> rt) {
15             this.el = el; left = lt; right = rt;
16         }
17     }
```



```

1  /* Cây BST tổng quát
2     Hầu hết các hành vi cần có trên cây BST đã được hiện thực trong lớp này
3  */
4  package trees;
5  import java.util.Stack;
6  public class BST<T extends Comparable<? super T>> {
7      protected BSTNode<T> root = null;
8      public BST() { }
9      public void clear() { root = null; }
10     public boolean isEmpty() { return root == null; }
11     /*Hàm chèn phần tử mới KHÔNG DÙNG KỸ THUẬT ĐỆ QUY. Phần tử mới el sẽ được
12     chèn thành một nút lá mới trong cây và là con của một nút lá previous ĐÃ CÓ
13     ==>cần biết nút previous để cập nhật tham chiếu đến nút con mới này */
14     public void insert(T el) {
15         if (root == null) { // Nếu cây trống thì el sẽ đưa vào nút gốc
16             root = new BSTNode<T>(el);
17             return;
18         }
19         // Nếu cây có rồi, tìm nút cha để chèn nút con
20         BSTNode<T> p = root, prev = null; // p: biến tạm để chạy trước
21         while (p != null) { // tìm vị trí chèn nút mới
22             prev = p;
23             if (p.el.compareTo(el) < 0) p = p.right;
24             else p = p.left;
25         }
26         // Nếu nút cha có data < data của el ==> chèn vào bên phải
27         if (prev.el.compareTo(el) < 0) prev.right = new BSTNode<T>(el);
28         // Ngược lại, chèn bên trái, chấp nhận nút trùng lặp
29         // Nếu không muốn trị trùng lặp thì thêm if (prev.el.compareTo(el) > 0)
30         else prev.left = new BSTNode<T>(el);
31     }
32     /* Chèn phần tử DÙNG KỸ THUẬT ĐỆ QUY- recursion. Nếu CÂY CON p trống thì
33     cấp phát mới và trả về nút mới này. Nếu p không trống thì đệ quy tác vụ
34     xuống nút con để chèn vào nút lá */
35     protected BSTNode<T> recInsert(BSTNode<T> p, T el) {
36         if (p == null) p = new BSTNode<T>(el);
37         else if (p.el.compareTo(el) < 0) p.right = recInsert(p.right, el);
38         // Nếu không muốn trị trùng lặp thì thêm if (prev.el.compareTo(el) > 0)
39         else p.left = recInsert(p.left, el);
40         return p;
41     }
42     // Chèn phần tử el vào cây
43     public void recInsert(T el) {
44         root = recInsert(root, el);
45     }

```

```

46 // vì data có thứ tự nên việc tìm kiếm được hiện thực dễ dàng nhờ vòng lặp
47 // O(logn) cho cây cân bằng, O(n) cho cây suy thoái về 1 hướng, n: số nút
48 protected T search(T el) {
49     BSTNode<T> p = root;
50     while (p != null)
51         if (el.equals(p.el)) return p.el;
52         else if (el.compareTo(p.el) < 0) p = p.left;
53         else p = p.right;
54     return null;
55 }
56 public boolean isInTree(T el) { // kiểm tra el có trong cây không
57     return search(el) != null;
58 }
59 // viếng nút p
60 protected void visit(BSTNode<T> p) {
61     // code phù hợp với việc xử lý của bài toán
62     System.out.print(p.el + " ");
63 }
64 // Các phép duyệt nút cơ bản
65 protected void inorder(BSTNode<T> p) {
66     if (p != null) {
67         inorder(p.left); // NODE
68         visit(p); // LEFT
69         inorder(p.right); // RIGHT
70     }
71 }
72 protected void preorder(BSTNode<T> p) {
73     if (p != null) {
74         visit(p); // NODE
75         preorder(p.left); // LEFT
76         preorder(p.right); // RIGHT
77     }
78 }
79 protected void postorder(BSTNode<T> p) {
80     if (p != null) {
81         postorder(p.left); // LEFT
82         postorder(p.right); // RIGHT
83         visit(p); // NODE
84     }
85 }
86 // Các tác vụ duyệt toàn bộ cây
87 public void preorder() { preorder(root); }
88 public void inorder() { inorder(root); }
89 public void postorder() { postorder(root); }
90 // Xóa phần tử el bằng PHƯƠNG PHÁP TRỘN
91 // Phương pháp này sẽ cập nhật các tham chiếu rồi xóa phần tử được chọn
92 // sao cho thứ tự dữ liệu phải được bảo tồn
93 // TÌNH HUỐNG NÚT BỊ XOÁ CÓ 2 CÂY CON, TRỘN 2 CÂY CON NÀY THÀNH 1
94 public void deleteByMerging(T el) {
95     // Đi tìm nút bị xóa p (ứng với el) và nút cha previous của nó
96     BSTNode<T> p = root, prev = null;
97     while (p != null && !p.el.equals(el)) {
98         prev = p;
99         if (p.el.compareTo(el) < 0) p = p.right;
100         else p = p.left;
101 }

```

```

102 // Khởi tạo nút cần bảo tồn, node, là nút bị xoá p
103 // sau đó cập nhật tuỳ tình huống
104 BSTNode<T> node = p;
105 BSTNode<T> tmp; // biến tạm, nút biên phải của cây con trái
106 if (p != null && p.el.equals(el)) { // có phần tử cần phải xoá
107     // Nếu node không có con phải => gắn con trái của nó vào nút cha
108     // => chuyển node về node.left để gắn vào previous sau này
109     if (node.right == null) node = node.left;
110     // Nếu node không có con trái => gắn con phải của nó vào nút cha
111     // => chuyển node về node.right để gắn vào previous sau này
112     else if (node.left == null) node = node.right;
113     else { // node có cả 2 con, trộn các cây con
114         // tìm nút biên phải của cây con trái
115         tmp = node.left;
116         while (tmp.right != null) tmp = tmp.right;
117         // Gắn bên phải của node vào bên phải của tmp vì
118         // chúng có trị lớn hơn
119         tmp.right = node.right;
120         // Giữ node.left để hiệu chỉnh vào previous
121         node = node.left;
122     }
123     // Trường hợp nút bị xoá là nút gốc
124     if (p == root) root = node; // node là gốc của cây kết quả
125     // nếu xoá nút con trái của nút cha
126     else if (prev.left == p) prev.left = node;
127     // nếu xoá nút con phải của nút cha
128     else prev.right = node;
129 }
130 else if (root != null)
131     System.out.println("Element " + el + " is not in the tree");
132 else System.out.println("the tree is empty");
133 }
134 // Xoá phần tử el dựa trên phép sao chép
135 // Copy dữ liệu của nút con BIÊN PHẢI của cây con trái vào nút bị xoá rồi
136 // huỷ nút biên phải này
137 public void deleteByCopying(T el) {
138     // Đi tìm nút bị xoá p (ứng với el) và nút cha previous của nó
139     BSTNode<T> p = root, prev = null;
140     while (p != null && !p.el.equals(el)) { // find the node p
141         prev = p; // with element el;
142         if (p.el.compareTo(el) < 0)
143             p = p.right;
144         else p = p.left;
145     }
146     // Khởi tạo nút cần bảo tồn là nút bị xoá sau đó cập nhật tuỳ tình huống
147     BSTNode<T> node = p;
148     if (p != null && p.el.equals(el)) { // có nút bị xoá
149         // node này không có con phải bảo tồn con trái
150         if (node.right == null) node = node.left;
151         // node này không có con trái bảo tồn con phải
152         else if (node.left == null) node = node.right;
153         else { // node cần xoá có cả 2 con
154             // Tìm tmp là nút biên phải của cây con trái của node
155             BSTNode<T> previous = node;
156             BSTNode<T> tmp = node.left;

```

```

157         while (tmp.right != null) {
158             previous = tmp; tmp = tmp.right;
159         }
160         // chép data của nút biên phải vào nút cần xoá node
161         node.el = tmp.el;
162         // Nút biên trái không có cây con phải -> cập nhật previous.left
163         if (previous == node) previous.left = tmp.left;
164         // Nút biên phải có cây con phải, previous nằm dưới node
165         // Móc temp.left( có trị lớn hơn) vào bên phải
166         // của previous (trị nhỏ hơn)
167         else previous.right = tmp.left;
168     }
169     // nếu nút bị xoá là nút gốc, thay gốc bằng node
170     if (p == root) root = node;
171     // nếu nút bị xoá là con trái của nút cha,
172     // thay con trái của previous
173     else if (prev.left == p) prev.left = node;
174     // nếu nút bị xoá là con phải của nút cha,
175     // thay con phải của previous
176     else prev.right = node;
177 }
178 else if (root != null) // không có nút bị xoá
179     System.out.println("Element " + el + " is not in the tree");
180 else System.out.println("the tree is empty");
181 }
182 // Duyệt preorder bằng phép lặp dùng Stack
183 public void iterativePreorder() {
184     BSTNode<T> p = root;
185     Stack<BSTNode<T>> stack = new Stack<BSTNode<T>>();
186     if (p != null) {
187         stack.push(p);
188         while (!stack.isEmpty()) {
189             p = stack.pop();
190             visit(p); // NODE
191             // Cắt right vào trước để lấy ra sau
192             if (p.right != null) stack.push(p.right);
193             // Cắt left vào sau để lấy ra trước
194             if (p.left != null) stack.push(p.left);
195         }
196     }
197 }
198 // Duyệt inorder bằng phép lặp dùng Stack
199 public void iterativeInorder() {
200     BSTNode<T> p = root;
201     Stack<BSTNode<T>> stack = new Stack<BSTNode<T>>();
202     while (p != null) {
203         while(p != null) {
204             // Cắt right vào stack trước để lấy ra sau
205             if (p.right != null) stack.push(p.right);
206             stack.push(p); // Cắt nút hiện hành sau để lấy ra trước right
207             p = p.left; // chuyển sang trái để duyệt left trước
208         }

```

```

209         p = stack.pop(); // Lấy ra 1 nút trong stack
210         // khi còn nút phải xét và nút này không có con phải
211         while (!stack.isEmpty() && p.right == null) {
212             visit(p);
213             p = stack.pop();
214         }
215         visit(p); // viếng nút có con phải
216         if (!stack.isEmpty()) p = stack.pop();
217         else p = null;
218     }
219 }
220 // Duyệt postorder dùng stack tự quản lý
221 // Cần 2 stack: stack để thao tác
222 // Giải thuật này đòi hỏi bộ nhớ phải đủ chứa toàn bộ các nút của cây
223 public void iterativePostorder2() {
224     BSTNode<T> p = root;
225     // stack giúp cất tạm các nút
226     Stack<BSTNode<T>> stack = new Stack<BSTNode<T>>();
227     // stack chứa thứ tự duyệt
228     Stack<BSTNode<T>> output = new Stack<BSTNode<T>>();
229     if (p != null) { // left-to-right postorder = right-to-left preorder;
230         stack.push(p);
231         while (!stack.isEmpty()) {
232             p = stack.pop(); // lấy ra NODE
233             output.push(p); // cất vào output
234             if (p.left != null) stack.push(p.left);
235             if (p.right != null) stack.push(p.right);
236         }
237         // duyệt từng nút theo thứ tự đã có
238         while (!output.isEmpty()) {
239             p = output.pop();
240             visit(p);
241         }
242     }
243 }
244 public void iterativePostorder() {
245     BSTNode<T> p = root, q = root;
246     Stack<BSTNode<T>> stack = new Stack<BSTNode<T>>();
247     while (p != null) {
248         // cất các left vào stack
249         for (; p.left != null; p = p.left) stack.push(p);
250         while (p != null && (p.right == null || p.right == q)) {
251             visit(p);
252             q = p;
253             if (stack.isEmpty())
254                 return;
255             p = stack.pop();
256         }
257         stack.push(p);
258         p = p.right;
259     }
260 }

```

```

261 public void breadthFirst() {
262     BSTNode<T> p = root;
263     MyQueue<BSTNode<T>> queue = new MyQueue<BSTNode<T>>();
264     if (p != null) {
265         queue.enqueue(p);
266         while (!queue.isEmpty()) {
267             p = queue.dequeue();
268             visit(p);
269             if (p.left != null)
270                 queue.enqueue(p.left);
271             if (p.right != null)
272                 queue.enqueue(p.right);
273         }
274     }
275 }
276 public void MorrisInorder() {
277     BSTNode<T> p = root, tmp;
278     while (p != null)
279         if (p.left == null) {
280             visit(p);
281             p = p.right;
282         }
283         else {
284             tmp = p.left;
285             while (tmp.right != null && // go to the rightmost node of
286                    tmp.right != p) // the left subtree or
287                 tmp = tmp.right; // to the temporary parent of p;
288             if (tmp.right == null) { // if 'true' rightmost node was
289                 tmp.right = p; // reached, make it a temporary
290                 p = p.left; // parent of the current root,
291             }
292             else { // else a temporary parent has been
293                 visit(p); // found; visit node p and then cut
294                 tmp.right = null; // the right pointer of the current
295                 p = p.right; // parent, whereby it ceases to be
296                 // a parent;
297             }
298         }
299 public void MorrisPreorder() {
300     BSTNode<T> p = root, tmp;
301     while (p != null) {
302         if (p.left == null) {
303             visit(p);
304             p = p.right;
305         }
306         else {
307             tmp = p.left;
308             while (tmp.right != null && // go to the rightmost node of
309                    tmp.right != p) // the left subtree or
310                 tmp = tmp.right; // to the temporary parent of p;
311             if (tmp.right == null) { // if 'true' rightmost node was
312                 visit(p); // reached, visit the root and
313                 tmp.right = p; // make the rightmost node a temporary
314                 p = p.left; // parent of the current root,
315             }

```

```

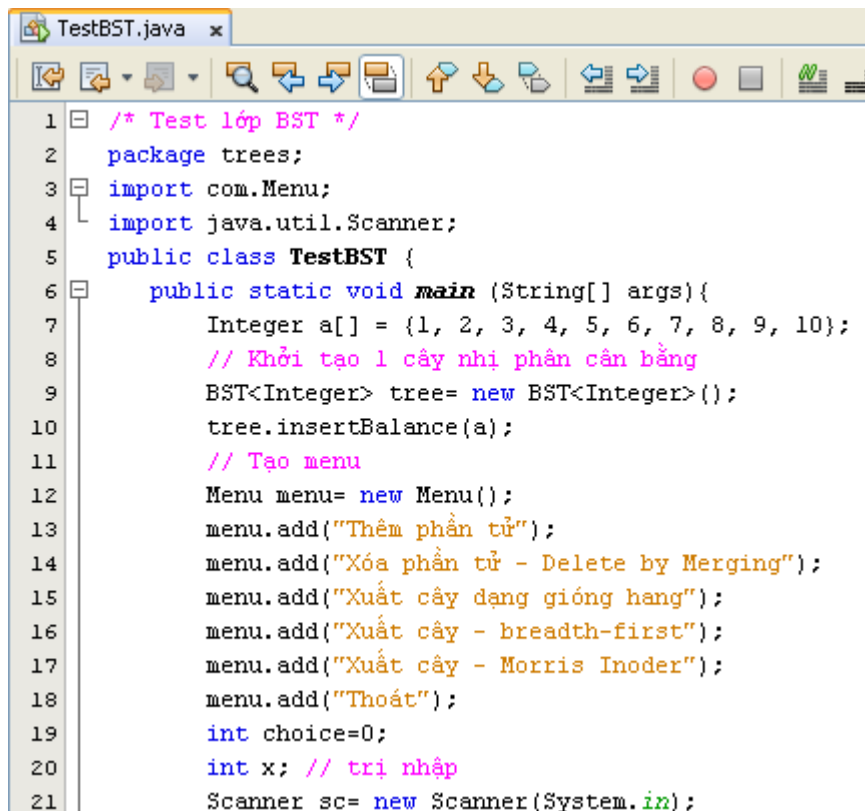
316         else { // else a temporary parent has been
317             tmp.right = null; // found; cut the right pointer of
318             p = p.right; // the current parent, whereby it ceases
319         } // to be a parent;
320     }
321 }
322 }
323 public void MorrisPostorder() {
324     BSTNode<T> p = new BSTNode<T>(), tmp, q, r, s;
325     p.left = root;
326     while (p != null)
327         if (p.left == null)
328             p = p.right;
329         else {
330             tmp = p.left;
331             while (tmp.right != null && // go to the rightmost node of
332                 tmp.right != p) // the left subtree or
333                 tmp = tmp.right; // to the temporary parent of p;
334             if (tmp.right == null) { // if 'true' rightmost node was
335                 tmp.right = p; // reached, make it a temporary
336                 p = p.left; // parent of the current root,
337             }
338             else { // else a temporary parent has been found;
339                 // process nodes between p.left (included) and p (excluded)
340                 // extended to the right in modified tree in reverse order;
341                 // the first loop descends this chain of nodes and reverses
342                 // right pointers; the second loop goes back, visits nodes,
343                 // and reverses right pointers again to restore the pointers
344                 // to their original setting;
345                 for (q = p.left, r = q.right, s = r.right;
346                     r != p; q = r, r = s, s = s.right)
347                     r.right = q;
348                 for (s = q.right; q != p.left;
349                     q.right = r, r = q, q = s, s = s.right)
350                     visit(q);
351                 visit(p.left); // visit node p.left and then cut
352                 tmp.right = null; // the right pointer of the current
353                 p = p.right; // parent, whereby it ceases to be
354             } // a parent;
355         }
356     }

```

```

357 // Duyệt inorder dạng giống hàng
358 private void visit_align(BSTNode p, int level){
359     if (p==null) return;
360     if (level>0){
361         for (int i=0;i<level-1;i++)System.out.print("  ");
362         System.out.print("|__");
363     }
364     System.out.println(p.el); // NODE
365     visit_align(p.left, level+1); // left
366     visit_align(p.right, level+1); // right
367 }
368 // Xuât cây dạng giống hàng
369 public void print_align (){
370     visit_align(root,0);
371 }
372 // chèn 1 nhóm trị DÃ CÓ THỨ TỰ TĂNG từ vị trí first đến vị trí last
373 // vào cây để tạo cây cân bằng
374 public void insertBalance(T data[], int first, int last) {
375     if (first <= last) {
376         int middle = (first + last)/2;
377         insert(data[middle]);
378         insertBalance(data,first,middle-1);
379         insertBalance(data,middle+1,last);
380     }
381 }
382 // chèn cả nhóm trị DÃ CÓ THỨ TỰ TĂNG vào cây để tạo cây cân bằng
383 public void insertBalance(T data[]) {
384     insertBalance(data,0,data.length-1);
385 }
386 }
387 }

```



```

TestBST.java x
1  /* Test lớp BST */
2  package trees;
3  import com.Menu;
4  import java.util.Scanner;
5  public class TestBST {
6      public static void main (String[] args){
7          Integer a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
8          // Khởi tạo 1 cây nhị phân cân bằng
9          BST<Integer> tree= new BST<Integer>();
10         tree.insertBalance(a);
11         // Tạo menu
12         Menu menu= new Menu();
13         menu.add("Thêm phần tử");
14         menu.add("Xóa phần tử - Delete by Merging");
15         menu.add("Xuất cây dạng giống hàng");
16         menu.add("Xuất cây - breadth-first");
17         menu.add("Xuất cây - Morris Inoder");
18         menu.add("Thoát");
19         int choice=0;
20         int x; // trị nhập
21         Scanner sc= new Scanner(System.in);

```



```

22         do {
23             choice= menu.getUserChoice();
24             switch(choice){
25                 case 1:
26                     System.out.print("Nhập trị cần thêm vào cây:");
27                     x= Integer.parseInt(sc.nextLine());
28                     tree.insert(x);
29                     System.out.print("Tri " + x + " đã được thêm vào cây:");
30                     break;
31                 case 2:
32                     System.out.print("Nhập trị cần xoá khỏi cây:");
33                     x= Integer.parseInt(sc.nextLine());
34                     tree.deleteByMerging(x);
35                     break;
36                 case 3: tree.print_align(); break;
37                 case 4: tree.breadthFirst();break;
38                 case 5: tree.MorrisInorder(); break;
39             }
40         }
41         while (choice>0 && choice <6);
42     }
43 }

```

### **Bài tập**

Sử dụng cây BST dựa trên mã sinh viên (code), viết chương trình quản lý danh sách học sinh <code, name, mark> có các tác vụ: thêm/xoá/sửa điểm học sinh.

**Gợi ý:** Xây dựng lớp Student có implements interface java.lang.Comparable, override hai hành vi:

```

public int compareTo(Object st) {
    return this.code.compareTo(((Student)st).code);
}
public String toString(){
    return code + "," + name + "," + mark;
}

```

**Hành vi compareTo(...)** được dùng để so sánh mã khi thêm sinh viên vào cây cũng như tìm kiếm sinh viên dựa trên mã học sinh.

**Hành vi toString()** được dùng khi xuất học sinh.