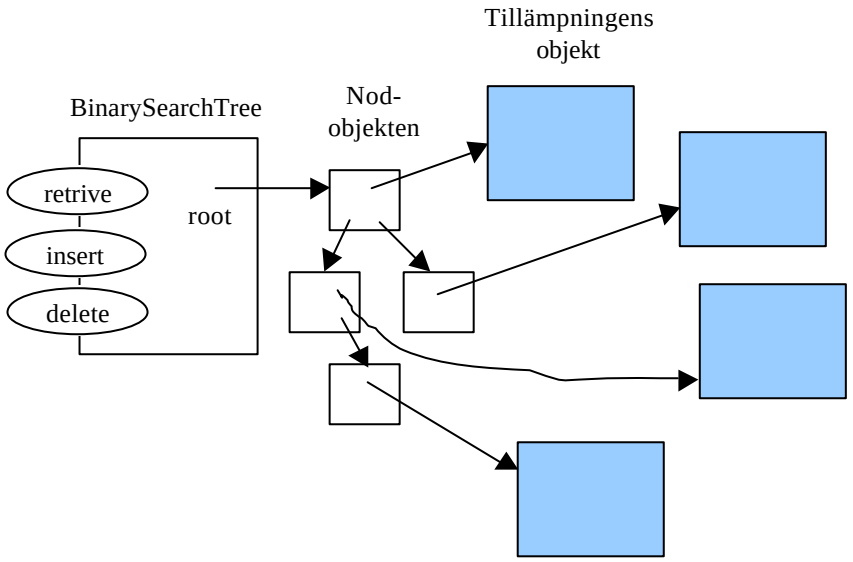
Jozef Swiatycki DSV

Några kommentarer om skillnader mellan klassen för binära sökträd från föreläsning 10 och den klass för binära sökträd som presenteras i Carranp & Prichard kap. 10

På föreläsning 10 visade jag ett binärt sökträd för en bestämd typ av objekt (klassen TNode med ett heltals-id och en sträng). Det binära sökträdet implementerades i klassen BinSearchTree. Denna klass är inte återanvändbar i den betydelsen att den inte kan användas för objekt av andra klasser än just TNode med de angivna datatyperna.

Om man ville gör trädklassen mer generell och återanvändbar så måste man separera data från nodobjekten på samma sätt som gjordes för andra länkade strukturer tidigare på kursen. Alltså att nodobjekten innehåller en referens som kan peka ut (nästan) vilket objekt som helst:



Det finns dock ett problem: för att kunna implementera de olika algoritmerna (`retrive`, `insert` och `delete`) måste vi kunna jämföra de okända objektens id-värden - hur skall vi kunna göra det utan att känna till objektens klass?

Vi behöver veta två saker om tillämpningens objekt: dels hur man får fram id-värdet, dels hur man jämför två sådana id-värden med varandra. När det gäller jämförelse av id-värden så finns det en förberedelse för det i Java: vi kan kräva att objektens id-värden uppfyller gränssnittet `Comparable` som innebär att id-värdet innehåller en metod med namnet `compareTo`. Denna metod tar ett annat objekt som argument och returnerar 0 om det egna objektet är lika med argumentobjektet, ett negativt värde om det egna objektet är mindre än argumentet och ett positivt värde om det egna objektet är större än argumentet. T.ex. har klasserna `String`, `Integer`, `Double` o.s.v. var sin sådan metod (och de uppfyller gränssnittet `Comparable`). Vill man exempelvis kontrollera om en strängvariabel (säg `str`) innehåller ett strängvärde som är mindre än (i alfabetisk ordning) än strängen "gurka" så skriver man

```
if (str.compareTo("gurka") < 0)
```

Om vi vet att id-värden uppfyller gränssnittet `Comparable` så kan vi alltså använda metoden `compareTo` för att jämföra värdena, oberoende av vad de är för något.

För att veta hur vi skall få fram id-värdet för ett objekt så kan vi kräva att alla objekt som skall kunna stoppas in i trädet måste ha en metod, säg `getKey`, som returnerar id-värdet. Metoden skall i så fall returnera ett `Comparable`-objekt (som alltså kan vara `String`, `Integer` o.s.v., eller även objekt av en klass som tillämpningen själv har deklarerat, förutsatt att denna klass uppfyller gränssnittet `Comparable`). Metodens huvud skall alltså se ut så här:

```
public Comparable getKey();
```

Att kräva detta av tillämpningens klasser kan göras på två sätt, båda sätten kräver att vi hittar på ett typnamn för dessa klasser – i boken har man valt namnet `KeyedItem`.

Det ena sättet är att man deklarerar ett gränssnitt med detta namn och med `getKey`-metoden så att tillämpningens klasser kan ange att de implementerar detta gränssnitt:

```
public interface KeyedItem {
   public Comparable getKey();
}
```

Om en tillämpning vill kunna stoppa in objekt av en klass Person som identifieras med namn och dessutom har ett attribut, säg, vikt så kan den deklarera klassen så här:

```
class Person implements KeyedItem{
   String namn;
   int vikt;

   public Person(String n, int v){
      namn = n;
      vikt = v;
   } // konstruktorn

   public Comparable getKey() {  // Det går bra att returnera en String som Comparable
      return namn;               // eftersom String implementerar detta gränssnitt
   } // getKey

   // Andra metoder för Person
} // Person
```

Det andra sättet, som används i boken, går ut på att man istället för ett gränssnitt skapar en klass med namnet `KeyedItem` där man deklarerar en referens till ett id-värde av typen `Comparable`, en konstruktor som tar ett sådant id-värde som argument och lagrar i sitt objekt samt metoden `getKey`:

```
public abstract class KeyedItem {
   private Comparable key;

   public KeyedItem(Comparable k) {
      key = k;
   } // konstruktor

   public Comparable getKey() {
      return key;
   } // getKey
} // KeyedItem
```

Att klassen deklareras med ordet `abstract` innebär att man inte kan skapa objekt av denna klass, klassen kan bara användas för att ärva från den. Tillämpningens klasser kan alltså ärva från denna klass och kan då betraktas som `KeyedItem`. Att `KeyedItem`:s konstruktor kräver ett id-värde innebär att tillämpningens klasser måste ange detta id-värde i sin konstruktor i ett super-anrop. Klassen `Person` från exemplet ovan skulle nu se ut så här:

```
class Person extends KeyedItem{
   int vikt;    // String namn finns inte längre, är ersatt av KeyedItem:s key

   public Person(String namn, int v){
      super(namn);  // Här anropas KeyedItem:s konstruktor
      vikt = v;
   } // konstruktorn

   // Metoden getKey ärvs från KeyedItem

   // Andra metoder för Person
} // Person
```

Av dessa två sätt att kräva att tillämpningens klasser har metoden `getKey` (gränssnitt resp. klass att ärva från) skulle jag personligen föredra gränssnitt, därför att det låter tillämpningens klasser ärva från andra klasser (man kan bara ärva från en klass, men man kan implementera flera gränssnitt). I boken väljer man lösningen med en

klass att ärva från därför att man då kan se till att id-värdet bara kan anges vid skapande av objektet men kan sedan endast avläsas, inte ändras (id-värdet för ett objekt som är insorterat i ett binärt sökträd, eller någon annan sorterad datasamling, får inte ändras, det skulle rubba datasamlingens struktur) – i klassen `KeyedItem` är attributet `key` privat och det finns ingen metod som ändrar `key`.

Med dessa förberedelser kan nu nodklassen för vårt binärt sökträd deklareras så här:

```
class TreeNode {
   KeyedItem item;
   TreeNode left, right;

   public TreeNode(KeyedItem newItem){
      item = newItem;
   }
}
```

Denna deklaration skiljer sig lite från bokens: dels har jag lämnat attributen `item`, `left` och `right` oskyddade (detta för att få enklare kod i metoderna, egentligen borde man inte göra så men jag återkommer till hur det skulle se ut med skyddade attribut), dels har jag deklarerat referensen till tillämpningens objekt som `KeyedItem`, medan man i boken deklarerar den som `Object` – detta beror på att jag bara skall använda denna nodklass för binära sökträd, där jag vet att tillämpningens objekt måste vara `KeyedItem`, medan man i boken deklarerar en `TreeNode`-klass för användning i olika typer av binära träd, som alltså inte nödvändigtvis är sorterade. I bokens klass för binära sökträd tvingas man konvertera `item`-referensen till `KeyedItem` varje gång man skall använda den, vilket jag slipper – till priset av att nodklassen är för restriktiv för användning i osorterade binära träd.

Om vi nu utgår från den klass för binära sökträd som presenterades på föreläsning 10 så skulle det bli följande skillnader:

- där det står `TNode` skall det istället stå `TreeNode`
- där ett id-värdet förekommer som argument skall det stå `Comparable` istället för `int`
- metoden `retrive` skall inte returnera `TreeNode` (`TreeNode`-objekten är inte intressanta för tillämpningen) utan det `KeyedItem`-objekt som pekas ut från `TreeNode`-objektet. Metoden skulle t.ex. kunna se ut så här:

  ```
  public KeyedItem retrive(Comparable sought){
     TreeNode tmp = retriveNode(root, sought);
     if (tmp == null)  // Det fanns inget objekt med detta id-värde
        return null;
     else              // Noden som pekar ut objektet hittades, returnera objektet
        return tmp.item;
  } // KeyedItem
  ```

- överallt där man gör jämförelser mellan id-värden måste dessa ersättas med anrop till `compareTo`-metoden, dessutom måste id-värdena hämtas från objekten genom anrop till `getKey`-metoden. T.ex. skulle `retriveNode`-metoden se ut så här:

  ```
  private TreeNode retriveNode(TreeNode node, Comparable sought){
     if (node == null)
        return null;
     else if (node.item.getKey().compareTo(sought) == 0)
        return node;
     else if (node.item.getKey().compareTo(sought) > 0)
        return retriveNode(node.left, sought);
     else
        return retriveNode(node.right, sought);
  } // retriveNode
  ```

- metoden `insert` tar nu emot ett helt `KeyedItem`-objekt, inte enskilda data:

  ```
  public void insert(KeyedItem newItem){
     TreeNode newNode = new TreeNode(newItem);
     root = insertNode(root, newNode);
  } // insert
  ```

- `print-`metoderna kan inte implementeras – detta eftersom vi inte vet så mycket om objekten och därför inte vet hur de skall skrivas ut. Istället får man skapa mer generella traverseringsmöjligheter som låter tillämpningen får referenser till objekten i trädet i tur och ordning och utföra önskade operationer på objekten.

Innan vi tittar på traversering kan vi kort nämna hur koden skulle förändras om vi skyddade attributen i `TreeNode`-klassen. Attributen `item`, `left` och `right` skulle alltså deklareras som `private` och vi skulle införa metoder för att avläsa dem (`KeyedItem getItem()`, `TreeNode getLeft()`, `TreeNode getRight()`) och metoder för att sätta dem (`void setItem(KeyedItem item)`, `void setLeft(TreeNode left)` och `void setRight(TreeNode right)`).

Detta gör att där vi förut skrev t.ex. `node.item` får vi nu skriva `node.getItem()`, t.ex.

```
if (node.getItem().getKey().compareTo(sought) == 0)
```

Vidare skulle tilldelningar ersättas med anrop av `set`-metoderna. Exempelvis skulle metoden `insertNode` se ut så här:

```
private TreeNode insertNode(TreeNode node, TreeNode newNode) {
   if (node == null)
      node = newNode;
   else if (node.getItem().getKey().compareTo(newNode.getItem().getKey()) > 0)
      node.setLeft(insertNode(node.getLeft(), newNode));
   else
      node.setRight(insertNode(node.getRight(), newNode));
   return node;
} // insertNode
```

Jämför gärna med metoden `insertNode` från föreläsningsexemplet – ovanstående kod följer principer för objektorienterad programmering, men är nog mer svårläst för ett otränat öga...

Slutligen en kort kommentar om traversering av objekten i ett sådant återanvändbart binärt träd. Som nämndes tidigare är tillämpningar som använder trädklassen ointresserade av `TreeNode`-objekten, de borde inte ens känna till dessa objekt. Detta leder dock till ett problem när tillämpningarna skall besöka objekten i ett träd för att utföra någon operation på dem: det är bara trädklassens metoder som vet hur man traverserar `TreeNode`-objekten, men det är bara tillämpningen som vet hur man utför operationer på varje tillämpningsobjekt.

En lösning på detta problem (lösningen presenteras i boken i avsnittet "Tree Traversals Using an Iterator" på sid. 444) är att låta tillämpningen få tillgång till objekten från trädet ett objekt i taget. För att göra detta inför man en ny klass med namnet `TreeIterator`. Tillämpningen kan skapa objekt av denna klass och då ange vilket träd-objekt denna iterator skall traversera. Ett objekt av iterator-klassen innehåller en kö, som fylls med referenser till objekten i trädet i önskad ordning (inorder, preorder eller postorder). `TreeIterator`-klassen innehåller metoderna `setInorder`, `setPreorder` och `setPostorder` som traverserar trädet i resp. ordning och fyller den interna kön med referenser till objekten i trädet. Vidare finns de två metoderna `hasNext()` som returnerar true om kön inte är tom och `next()` som returnerar referensen till nästa objekt i kön.

Om en tillämpning har skapat ett binärt sökträd, t.ex. så här

```
BinarySearchTree personer = new BinarySearchTree();
```

och sedan skapat ett antal Person-objekt (av klassen i tidigare exempel) och stoppat in dem i trädet, t.ex. så här

```
personer.insert(new Person("Jozef", 73));
personer.insert(new Person("Stefan", 97));
```

så kan man nu skriva ut information om alla personer i sorteringsordning genom följande kod:

```
TreeIterator iter = new TreeIterator(personer);
iter.setInorder();

while (iter.hasNext()) {
   Person p = (Person)iter.next();
   System.out.println("Namn: " + p.getKey() + " vikt: " + p.vikt);
} // while
```