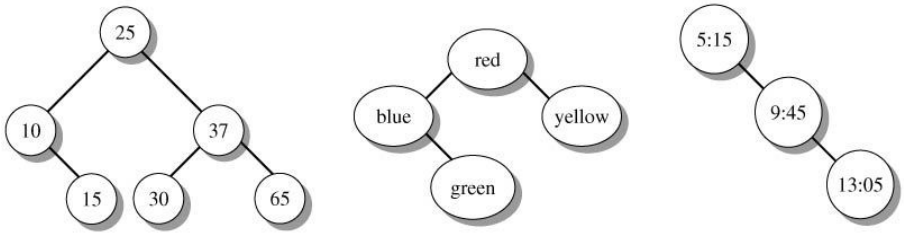
Binary Search Tree

Binary Search Tree

- Sebuah node di Binary Search Tree memiliki path yang unik dari root menurut aturan ordering
  - Sebuah Node, mempunyai subtree kiri yang memiliki nilai lebih kecil dari node tsb dan subtree kanan memiliki nilai lebih besar dari node tsb.
  - Tidak diperbolehkan ada node yang memiliki nilai yang sama.

Binary Search Tree



Binary Search Tree 1 (Integer objects)

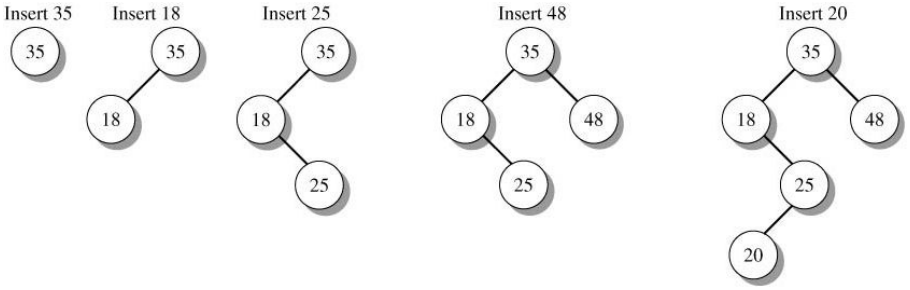
Binary Search Tree 2 (String objects)

Binary Search Tree 3 (Time24 objects)

Membangun Binary Search Tree

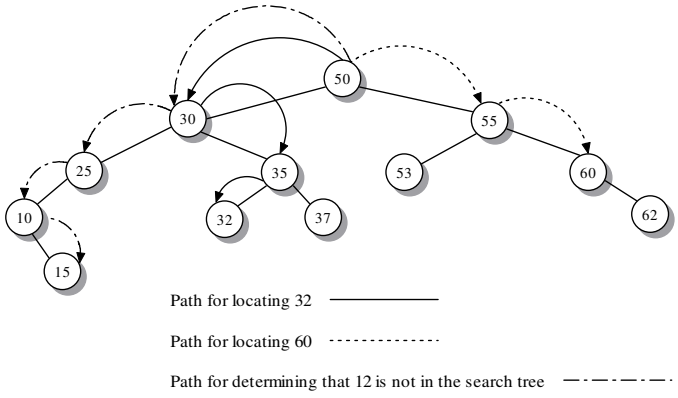
- Jika value dari node baru sama dengan value dari current node, maka mengembalikan nilai false.
- Jika value dari node baru kurang dari value dari current node maka :
  1) Jika anak kiri current node tidak null, maka ubah current node ke anak kiri tersebut, lakukan langkah 1.
  2) Jika anak kiri current node adalah null, maka tambahkan node baru tersebut sebagai anak kiri dari current node
- Jika value dari node baru lebih besar dari value dari current node maka :
  1. Jika anak kanan current node tidak null, maka ubah current node ke anak kanan tersebut, lakukan langkah 1.
  2. Jika anak kanan current node adalah null, maka tambahkan node baru tersebut sebagai anak kanan dari current node

Membangun Binary Search Tree



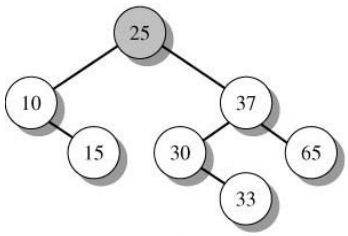
Menemukan Sebuah Node pada Binary Search Tree

- Pencarian sebuah node mempunyai proses yang sama dengan penyisipan



Menghapus Node Binary Search Tree

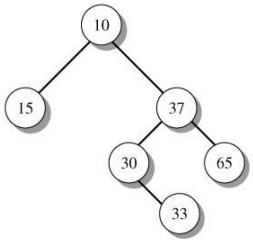
- Node root tidak dapat dihapus karena subtree kiri dan kanan akan menyebabkan tidak mempunyai parent, tapi nilai dari node bisa diganti dengan nilai yang lain.



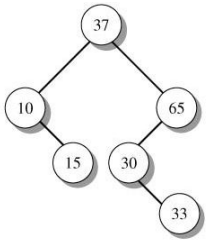
Delete root node 25

Menghapus Node Binary Search Tree

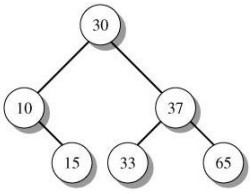
- Untuk menghapus sebuah node dari Binary Search Tree harus mengganti dengan node yang lain.



(a) Bad Solution:
Select 10 as the replacement value
15 is out of place



(b) Bad Solution:
Select 37 as the replacement value
30 and 33 are out of place



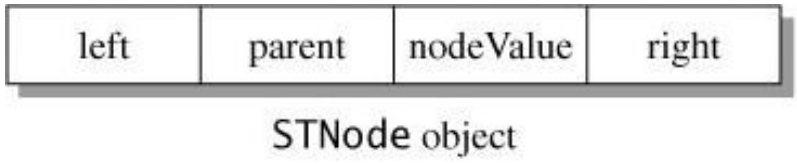
(c) Good Solution:
Select 30 as the replacement value
The search ordering is maintained

Class BinarySearchTree

- Tambahkan pada Class Binary Search Tree method:
  - add() : menyisipkan node baru, return true jika berhasil menambahkan node baru ke Tree, return false jika ada duplikat data
  - toString() : mengembalikan berupa String dengan algoritma inorder.
  - first() : mengembalikan nilai terkecil dari tree
  - last() : mengembalikan nilai terbesar dari tree.
  - find(): mencari sebuah nilai pada tree.

Class STNode

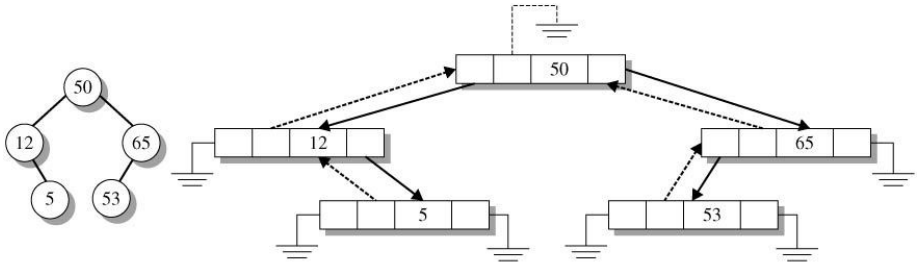
- Class STNode berisi informasi
  - nodeValue : menyimpan nilai dari node (T)
  - Left : menyimpan alamat dari anak kiri (STNode)
  - Right : menyimpan alamat dari anak kanan (STNode)
  - Parent : menyimpan alamat dari parent node tersebut (STNode)



STNode object

Implementing the STree Class (continued)

- - - - : link parent



Binary Search Tree

STnode Representation of Binary Search Tree using Parent Pointers

STree Class Private Members and Constructor

- Private methods findNode() and removeNode promote code reuse.

```
public class STree<T> implements Collection<T>,
Iterable<T>
{
   // reference to tree root
   private STNode<T> root;

   // number of elements in the tree
   private int treeSize;

   // increases whenever the tree changes; used
   // by an iterator to verify that it is in a
   // consistent state
   private int modCount;
```

STree Class Private Members and Constructor (continued)

```
// create an instance representing an empty
// tree; the root is null and the variables
// treeSize and modCount are initially 0
public STree()
{
   root = null;
   modCount = 0;
   treeSize = 0;
}
```

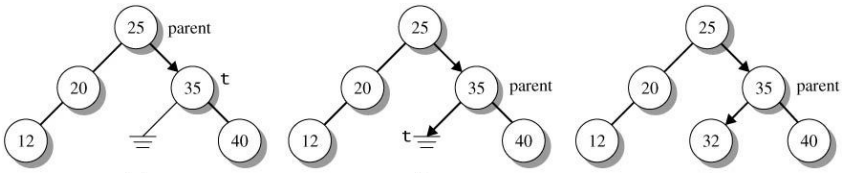

STree Class Private Members and Constructor (concluded)

```
   // iteratively traverse a path from the root to
   // the node whose value is item; return a reference
   // to the node containing item or null if the search
   // fails
   private STNode<T> findNode(Object item)
   { . . . }

   // private method used by remove() and the iterator
   // remove() to delete a node
   private void removeNode(STNode<T> dNode)
   { . . . }
}
```

Inserting and Locating a Node

- Method add() untuk menambahkan node baru pada Binary Search Tree
- Method ini berhasil jika dapat menambahkan node baru, mengembalikan nilai true, jika ada node yang sama mengembalikan nilai false



Adding element 32 to the binary search tree. Arrows indicate the path.

Method add()

```
// if item is not in the tree, insert it and
// return true; if item is a duplicate, do not
// insert it and return false
public boolean add(T item)
{
   // t is current node in traversal, parent the
   // previous node
   STNode<T> t = root, parent = null, newNode;
   int orderValue = 0;

   // terminate on an empty subtree
   while(t != null)
   {
      // update the parent reference
      parent = t;

      // compare item and the current node value
      orderValue = ((Comparable<T>)item).compareTo(
            t.nodeValue);
```

Method add()

```
      // if a match occurs, return false;
      // otherwise, go left or go right
      // following search tree order
      if (orderValue == 0)
         return false;   // exit, item not added
      else if (orderValue < 0)
         t = t.left;
      else
         t = t.right;
   }

   // create the new node
   newNode = new STNode<T>(item,parent);
```


Method add()

```
   if (parent == null)
      // this is the first node added; make it root
      root = newNode;
   else if (orderValue < 0)
      // attach newNode as the left child of parent
      parent.left = newNode;
   else
      // attach newNode as the right child of parent
      parent.right = newNode;

   // increment the tree size and modCount
   treeSize++;
   modCount++;

   // we added a node to the tree
   return true;
}
```

Method find()

- find() digunakan untuk mencari node dengan nilai tertentu.
- Jika berhasil maka mengembalikan nilai dari node tersebut, tapi jika tidak berhasil maka mengembalikan nilai null.

```
// search for item in the tree and return a
// reference to its value or null if item
// is not in the tree
public T find(T item)
{
   STNode<T> t = findNode(item);
   T value = null;

   if (t != null)
      value = t.nodeValue;

   return value;
}
```


Deleting a Node

- The private method removeNode() erases a node from the tree by finding a replacement node somewhere else in the tree and using it as a substitute for the deleted node.
  - Choose the node so that, when it takes the place of the deleted node, its value maintains the structure of the tree.
  - Subtrees for the deleted node and the replacement node must be reconnected in such a way that the new tree maintains search tree ordering.

Deleting a Node (continued)

- Notation:
  - dNode identifies the deleted node D. pNode, identifies the parent P of the deleted node. When pNode is null, we are deleting the root. The removeNode() method sets out to find a replacement node R with reference rNode.

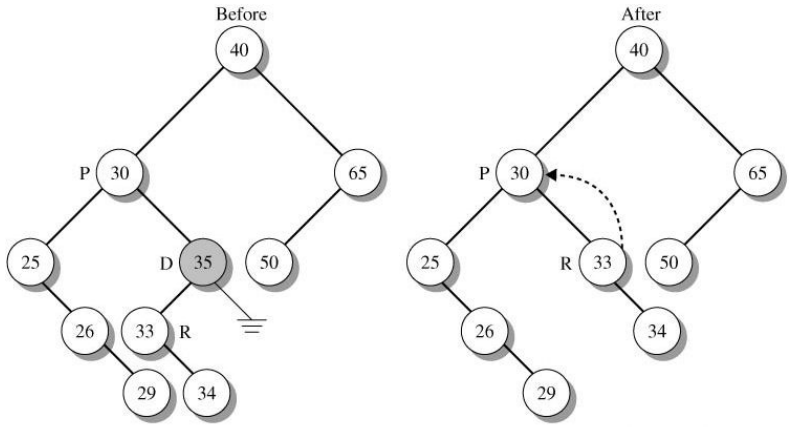
Deleting a Node (continued)

- The algorithm for finding a replacement node considers two cases that depend on the number of children attached to node D.

Deleted Node has an Empty Subtree

- The other child becomes the replacement node R. If the deleted node is a leaf node, it has two null children. In this case, the other node R is null.

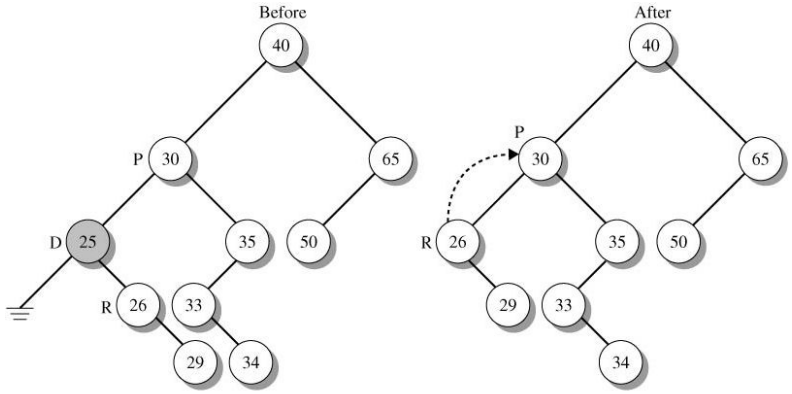
Deleted Node has an Empty Subtree (continued)



Delete node 35 with only a left child: Node R is the left child.

Attach node R as right child of P.

Deleted Node has an Empty Subtree (continued)



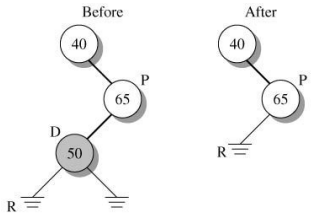
Delete node 25 with only a right child: Node R is the right child.

Attach node R as left child of P.

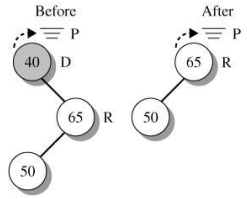
Deleted Node has an Empty Subtree (continued)

- The algorithm must handle two special cases, one in which the deleted node is a leaf node and the other in which the deleted node is the root.

Deleted Node has an Empty Subtree (continued)



(a) Delete leaf node 50 Replacement node is null



(b) Delete root node 40 Replacement node is the new root

Deleting a leaf node and a root node with a null child.

removeNode() (one null child)

```
private void removeNode(STNode<T> dNode)
{
   if (dNode == null)
      return;

   // dNode = reference to node D that is deleted
   // pNode = reference to parent P of node D
   // rNode = reference to node R that replaces D
   STNode<T> pNode, rNode;

   // assign pNode as a reference to P
   pNode = dNode.parent;
```

removeNode() (one null child continued)

```
   // if D has a null child, the
   // replacement node is the other child
   if (dNode.left == null || dNode.right == null)
   {
      if (dNode.right == null)
         rNode = dNode.left;
      else
         rNode = dNode.right;

      if (rNode != null)
         // the parent of R is now the parent of D
         rNode.parent = pNode;
```


removeNode() (one null child concluded)

```
      // deleting the root node; assign new root
      if (pNode == null)
         root = rNode;
      // attach R to the correct branch of P
      else if (((Comparable<T>)dNode.nodeValue).
            compareTo(pNode.nodeValue) < 0)
         pNode.left = rNode;
      else
         pNode.right = rNode;
   }
   ...
}
```