ELEC 372

Week 3

Administrative Notes

- Midterm date has been changed to Feb 12, 6:00-7:00 PM due to scheduling conflict
- Recall assignment 1 due Monday January 29 (in one week)

Abstract Data Types

- The classes we have been creating are examples of *abstract data types (ADT's)*
- ADT's represent a type that
  - provide high-level operations
  - can be instantiated any number of times
- E.g., A program may have many instances of
  - *TimeVal* values
  - *CourseList* lists of courses

Abstract Data Types

- Standard ADT's include
  - List
  - Tree
  - Graph
  - Stack
  - Queue
  - …etc
- Class libraries usually include standard implementations of these ADT's
  - E.g., Java library, C++ Standard Template Library, Borland C++ library, …

Example: Student Information ADT

- Student information is typically accessed via a student number
- In a student information database, therefore want operations allowing students to be found based on student number
- An ADT providing such a lookup service is called a *dictionary*

Student Information Dictionary

```
public interface StudentDictionary {
    // Add a new student to the dictionary
    public void addStudent (Student s);

    // Remove given student from dictionary
    public void removeStudent (Student s);

    // Look up a student by student number
    public Student findStudent (int studentNumber);

    // Print contents of dictionary to display
    // (for debugging)
    public void dump ();
}
```

Good Interface Description

- It should be possible to understand an interface from its description
  - I.e., a programmer should be able to use an ADT without reading its code
- Is this true here?

Adding a student

```
// Add a new student to the dictionary
public void addStudent (Student s);
```

- What if student 's' is already in the dictionary?
- What if a student with the same student number as 's' is already in the dictionary?
- What if s==null?
- What if the dictionary is full?

What does our program do?

```
// Add a new student to the dictionary
public void addStudent (Student s);
```

- What if student 's' is already in the dictionary?
  - *Operation has no effect*
- What if a student with the same student number as 's' is already in the dictionary?
  - *The earlier entry is replaced by the new entry*
- What if s==null?
  - *This is an error - program behaviour unspecified*
- What if the dictionary is full?
  - *Program may crash (ouch!! - better to provide a return value indicating success)*

Possible Implementations for Student Dictionary

- Array/Vector
- Linked List
- Binary Search Tree
- Hash Table
- …

Properties of Implementations

```
// Look up a student by student number
public Student findStudent (int studentNumber);
```

- Array/linked list implementations
  - Dictionary lookup is slow
    - E.g., at Queen's University, there are approximately 12,000 students currently enrolled
    - To find a student by student number, may need to search all 12,000 entries
    - Gets worse if we include students no longer enrolled
  - Specifically, dictionary lookup is *O(n),* where *n* represents the size of the dictionary (i.e., number of students)
    - *Recall:* An algorithm is *O(e)* for expression *e* involving *n* if there exists some constant *c* such that as *n* grows, the number of operations in the algorithm is bounded by *c*e*

Properties of Implementations

- We shall see how we can improve this by using a *binary search tree (BST)*
  - If the tree is balanced, a search is $O(\log_2 n)$

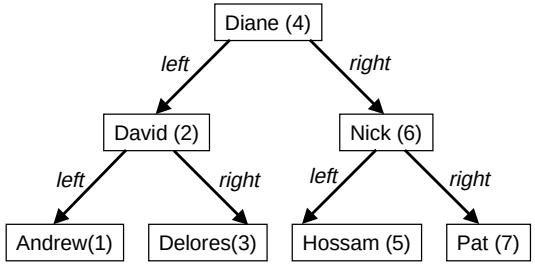
n	$\log_2 n$
1,000	10
1,000,000	20
1,000,000,000	30

Trees

Student Number	*Name*
1	Andrew
2	David
3	Delores
4	Diane
5	Hossam
6	Nick
7	Pat

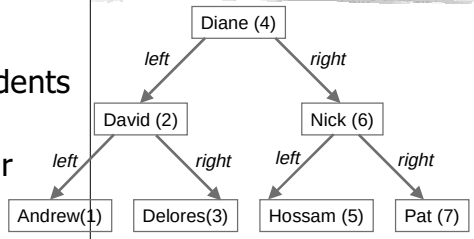
- Assume the dictionary contains these students
- Note it is accidental that the students names are alphabetical - this will make our drawings simpler, but in general is not true

Trees

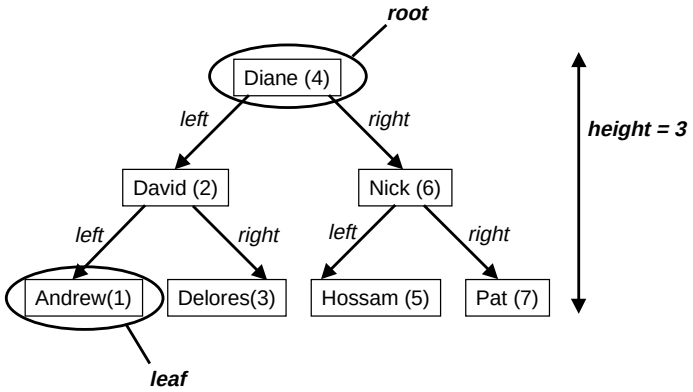


Trees

- A tree consists of a set of nodes
  - Here, nodes represent students (name, student number)
  - Each node has a set of 0 or more *children*
  - The maximum number of children in a tree is the trees *arity*
  - A binary tree has arity 2
  - The children of a binary tree are called the *left* and *right* child



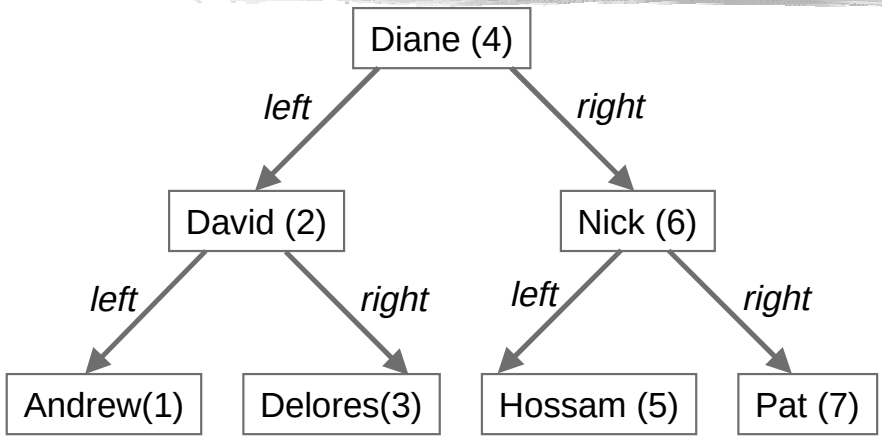
Trees



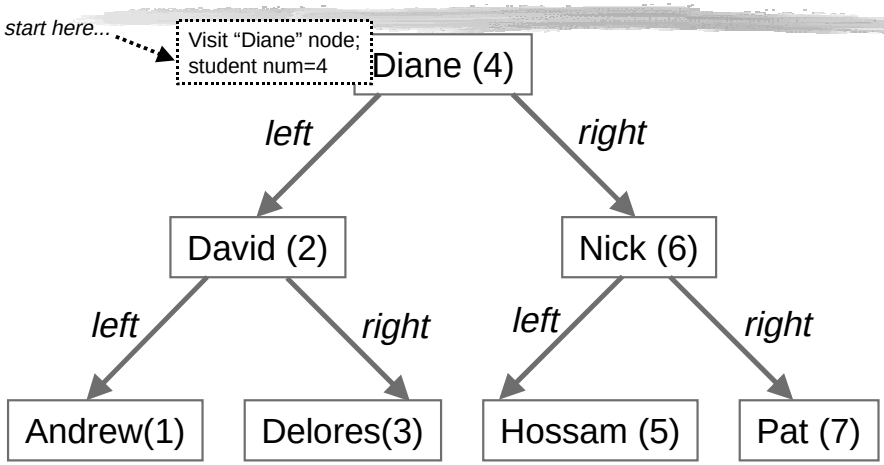
Tree Search

- Given a student number *sn*, find the corresponding name
  - Start at root of tree
    - If this node has student number *sn*
      - Return this node's name
    - else
      - Recursively search the left subtree for *sn*
      - Recursively search the right subtree for *sn*

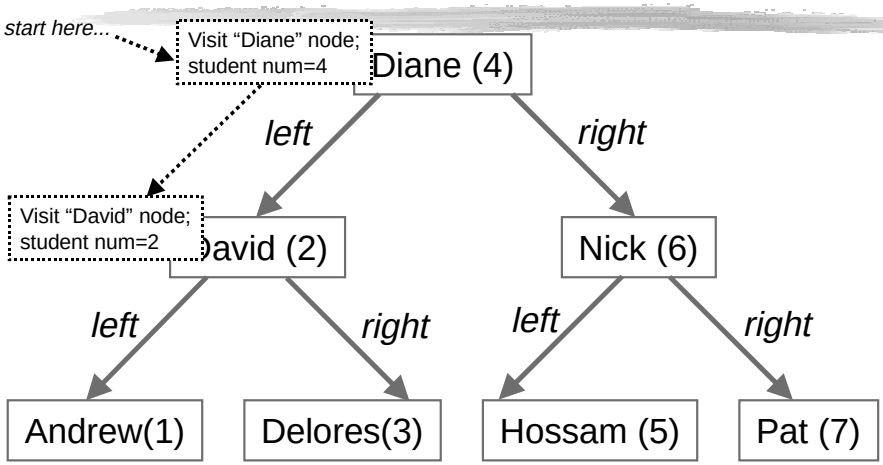
Eg, search for *sn*=3



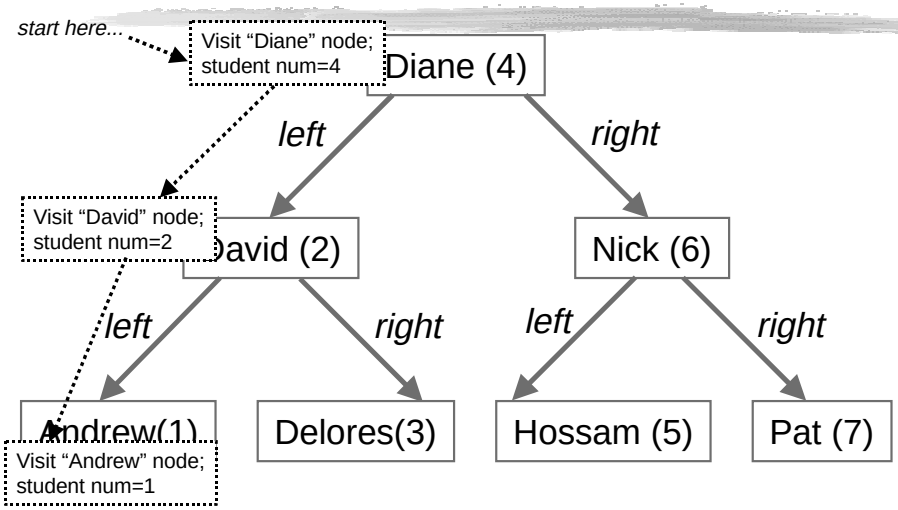
Eg, search for *sn*=3



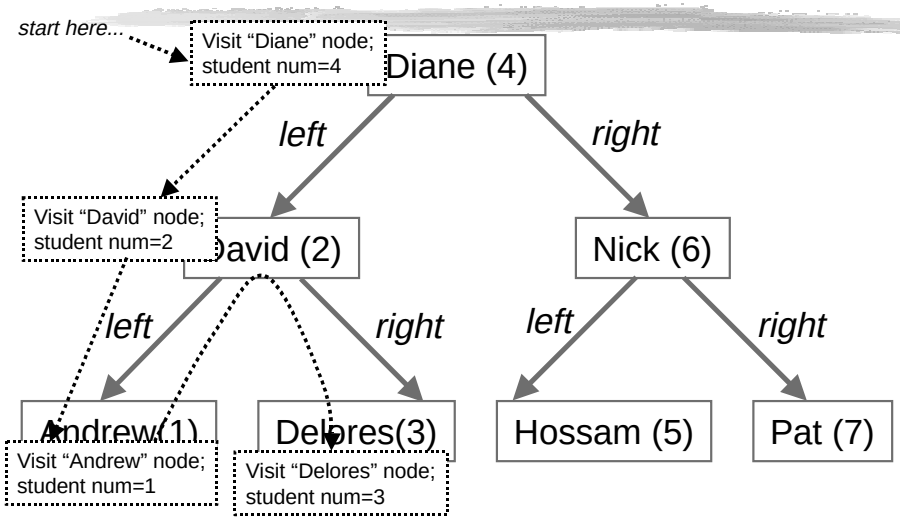
Eg, search for *sn*=3



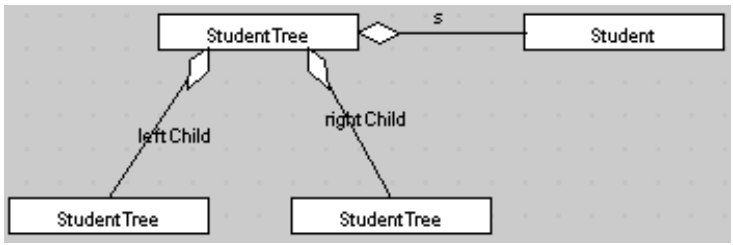
Eg, search for *sn*=3



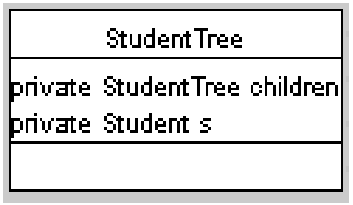
Eg, search for *sn*=3



UML Description of Tree



Refining Description



First Implementation

```
public class StudentTree {
    private Student s;

    public static final int leftChild = 0;
    public static final int rightChild = 1;
    private StudentTree [] children = new StudentTree [2];

    public StudentTree getChild (int n) {...}

    public void setChild (int n, StudentTree t) {...}

    public void setStudent (Student s) {...}

    public Student getStudent () {...}

    public Student findStudent (int studentNumber) {...}
}
```

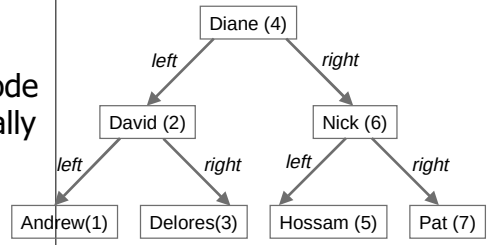
```
class StudentTree {
   private Student s;
   private StudentTree [] children;
   ...
   public Student findStudent (int studentNumber) {
      // check if the student is this node
      if (s.getStudentNumber() = studentNumber) {
         return s;
      } else {
         // search for student in any children
         for (int i=0; i<children.length; i++) {
            if (children[i] != null) {
               Student foundStudent =
                  children[i].findStudent (studentNumber);
               if (foundStudent != null) {
                  return foundStudent;
               }
            }
         }
      }
      // Student was not found
      return null;
   }
}
```

Complexity of Searching in a Tree

- Tree search is *O(n)*
  - In worst case, need to visit all *n* nodes
- Can improve with binary search tree

Binary Search Tree (BST)

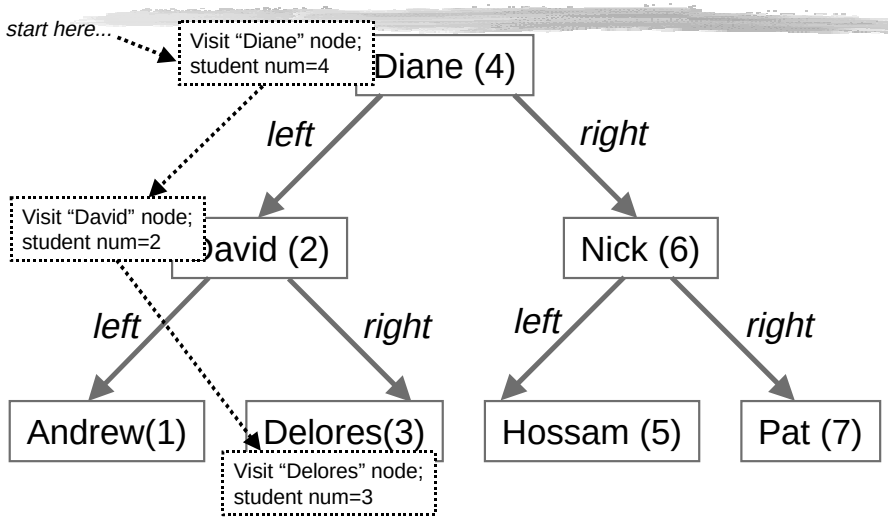
- Arrange tree with property that for any node
  - All elements to *left* of node *precede* that node numerically
  - All elements to *right* of node *follow* that node numerically
  - E.g., student numbers {1,2,3} appear to left of "Diane" node; student numbers {5,6,7} appear to right of "Diane" node



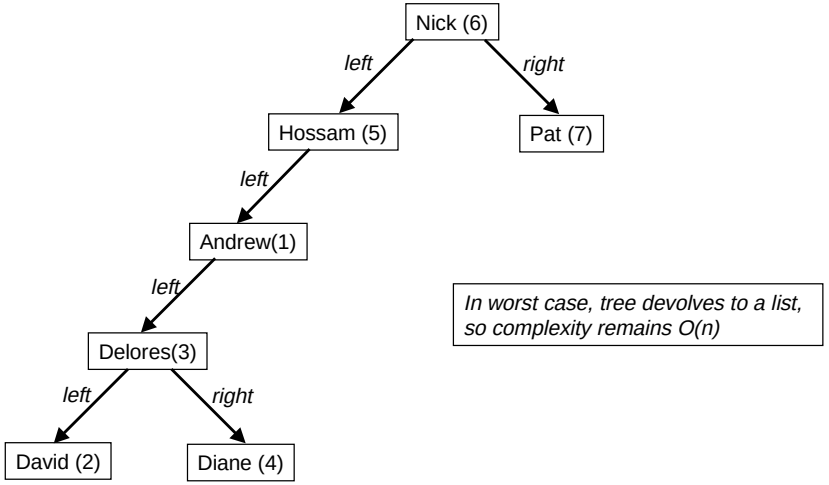
BST Search

- Given a student number *sn*, find the corresponding name
  - Start at root of tree
    - If this node has student number *sn*
      - Return this node's name
    - else if sn < this nodes student number
      - Recursively search the left subtree for *sn*
    - else (must be that sn > this nodes student number)
      - Recursively search the right subtree for *sn*

Eg, search for *sn*=3



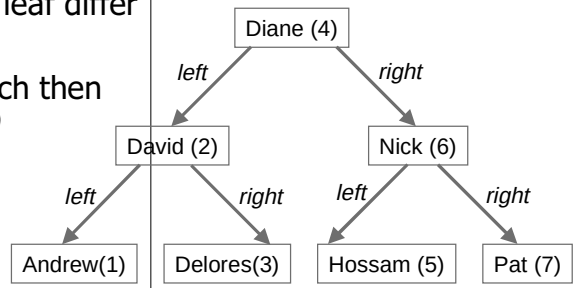
Complexity of BST Search



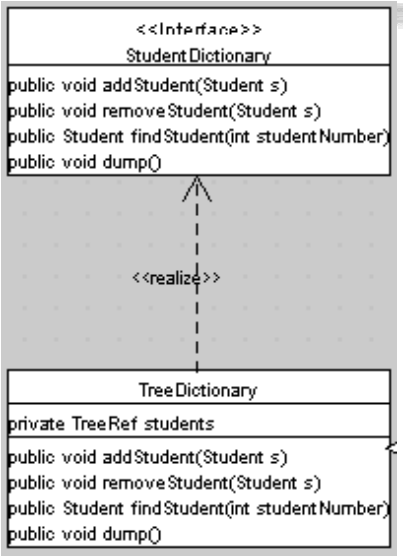
In worst case, tree devolves to a list, so complexity remains O(n)

AVL Tree

- Balanced binary search trees are called AVL trees
  - The lengths of any two paths from root to leaf differ by no more than 1
  - Complexity of search then becomes $O(\log_2 n)$

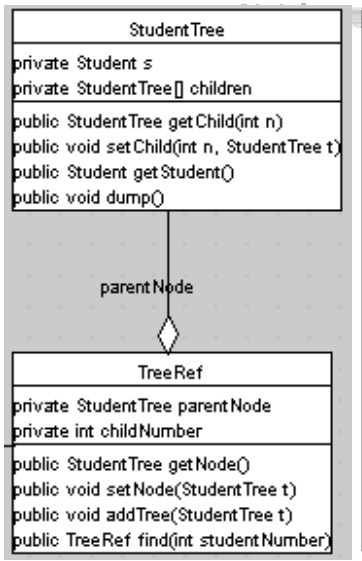


Implementing the Student Dictionary as a BST



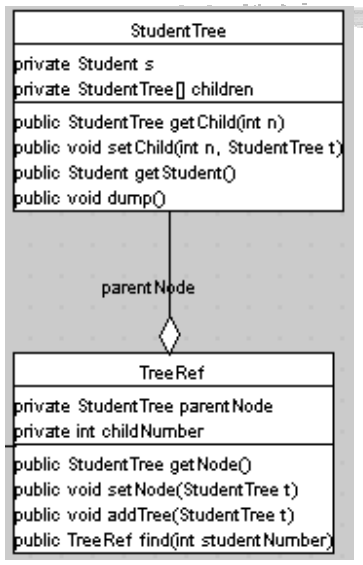
- *StudentDictionary* interface defines dictionary operations
- *TreeDictionary* implements these operations in terms of a binary search tree
  - *students* implemented via a *TreeRef* object

Implementing the Student Dictionary as a BST



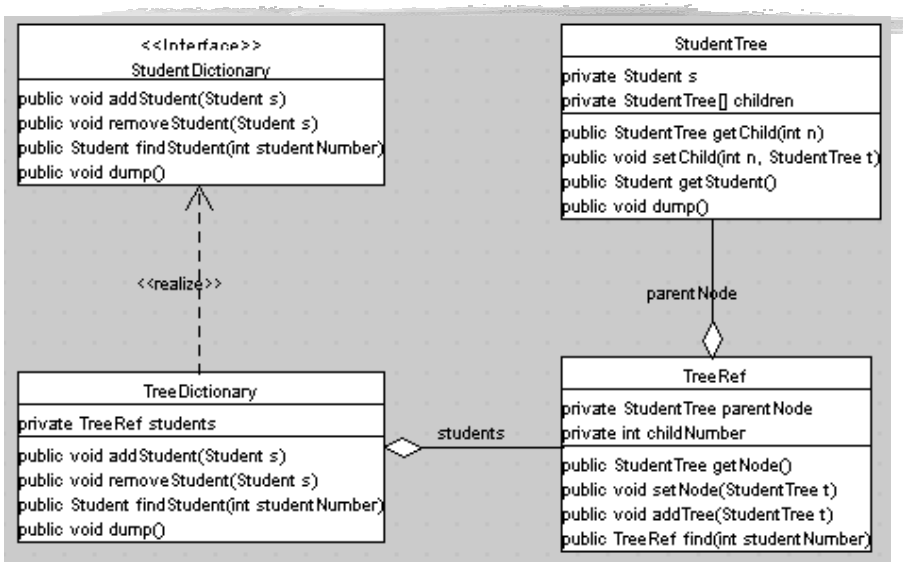
- *StudentTree*
  - implements the tree itself
  - *children* implemented as array of two elements
    - *leftChild* = 0
    - *rightChild* = 1
  - contains a student element *s*
  - operations
    - *getChild:* return left/right child
    - *setChild:* set value of left/right child
    - *getStudent:* get student associated with this node
    - *dump:* display contents of tree

Implementing the Student Dictionary as a BST

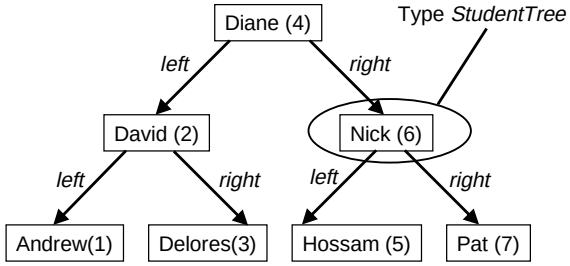


- *TreeRef*
  - implements mechanisms to traverse tree, modify tree structure
  - provides reference to a node via
    - a reference to its parent
    - the child number of the node (*leftChild* or *rightChild*)
  - contains a student element *s*
  - operations
    - *getNode:* return left/right child
    - *setNode:* set value of left/right child
    - *addTree:* get student associated with this node
    - *deleteNode:* deletes node from tree
    - *find:* find the student with the given student number

Implementing the Student Dictionary as a BST



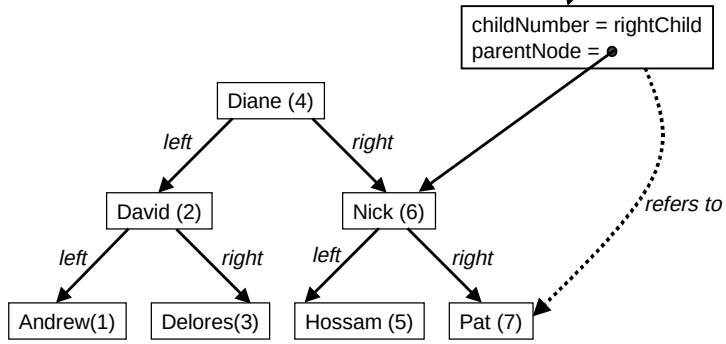
TreeRef's



Each node in the tree is of type StudentTree
- *contains a Student object (name, student number)*
- *contains references to its children (0..2)*

TreeRef's

Node of type TreeRef, referring to "Pat" node
- *I.e., the "Pat" node is the right child of "Nick"*



Searching

```
class TreeRef {
    ...
    TreeRef find (int studentNumber) {...}
    ...
}
```

- New version of *find* based on TreeRef
  - Invoke on reference to root of tree
  - Returns reference to tree node with this student
- Seems more complicated than last version!
  - Will help with implementation of *addTree*, *deleteNode*

In class TreeRef
- *Invoke with reference to root*

```
public TreeRef find (int studentNumber) {
       StudentTree t = getNode();

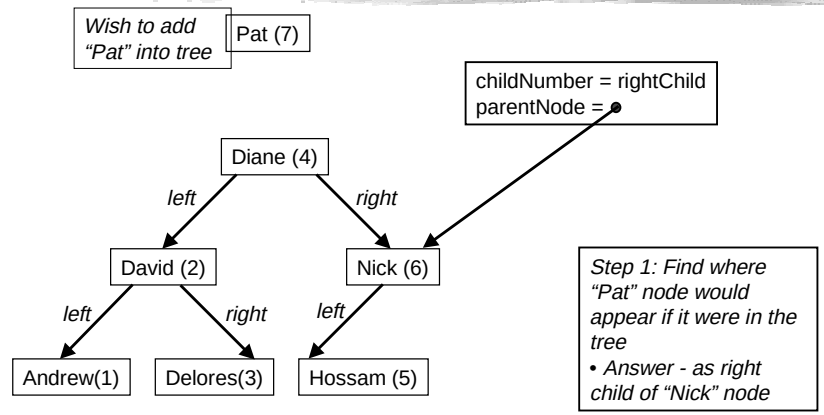
       if (t == null) {
           // No children - therefore cannot find this student
           return this;
       } else if (t.getStudent().getStudentNumber() == studentNumber) {
           // This node represents the student - return the
           // tree reference to this node
           return this;

       } else if (studentNumber < t.getStudent().getStudentNumber()) {
           // The student must be in the left subtree
           // Create a tree reference for the left child
           TreeRef leftRef
               = new TreeRef (t, StudentTree.leftChild);
           return leftRef.find (studentNumber);
       } else {
           // The student must be in the right subtree
           // Create a tree reference for the right child
           TreeRef rightRef
               = new TreeRef (t, StudentTree.rightChild);
           return rightRef.find (studentNumber);
       }
   }
```

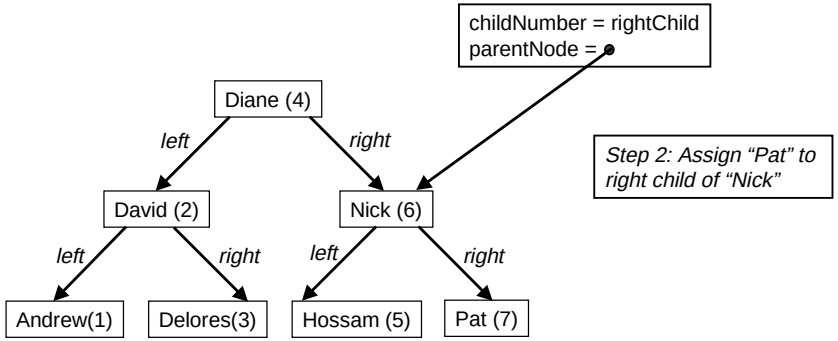
Adding a new node to the tree

- Find a reference to where that node would be found if it were in the tree
- Assign the node to that position in the parent

Adding a Node



Adding a Node



Adding a Node to a Tree

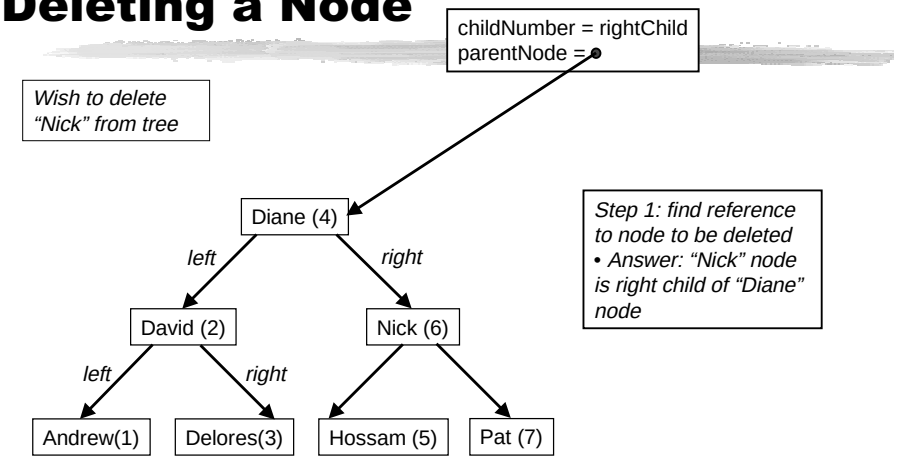
```
class TreeRef {
   ...
   public void addTree (StudentTree t) {
      // Traverse the tree to find the location to insert t.
      // Do this by finding the tree ref to the parent where
      // this tree should be inserted.
      int studentNumber = t.getStudent().getStudentNumber();
      TreeRef r = find (studentNumber);   // step 1
      r.setNode (t);   // step 2
   }
   ...
}
```

Note: doesn't keep tree balanced! Can you construct an example showing why not?

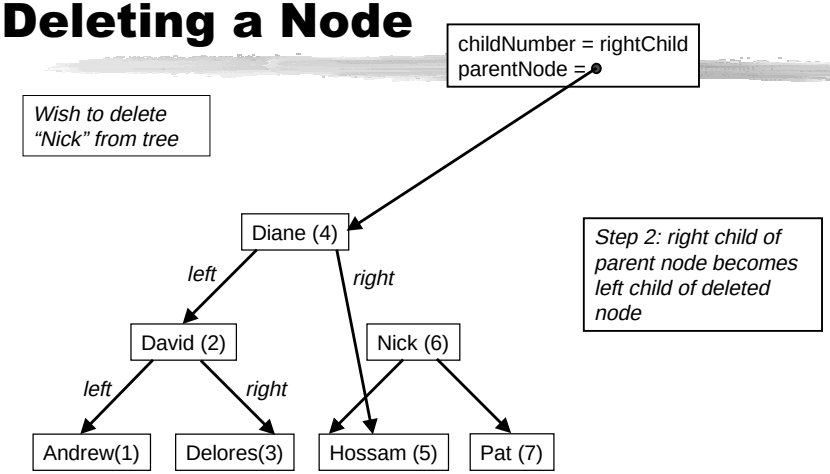
Deleting a node from the tree

- Find a reference to the node in the tree
- Remove that node from the tree
- Find appropriate places for the node's children
  - E.g., simple approach:
    - Put left subtree where deleted node was
    - Add right subtree into left subtree
    - …This approach works, but leads to very imbalanced trees

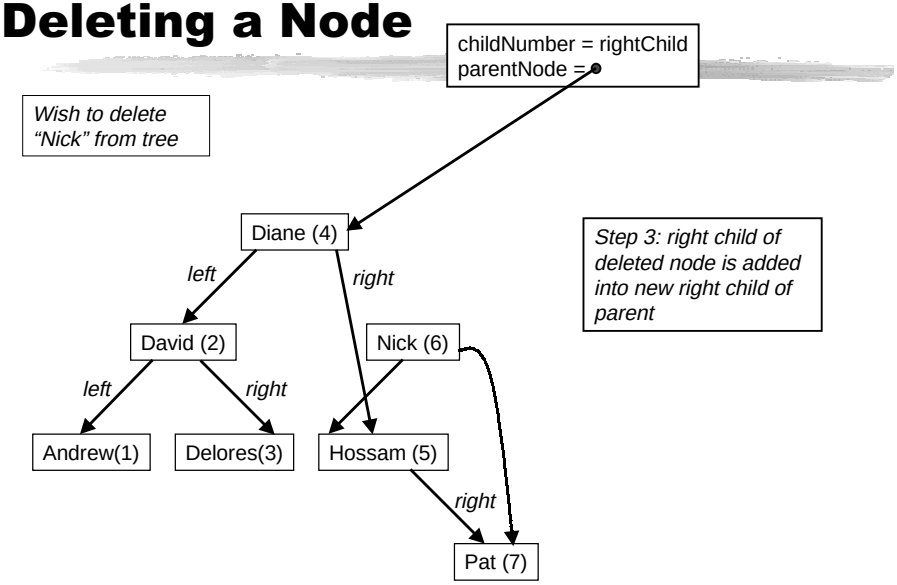
Deleting a Node



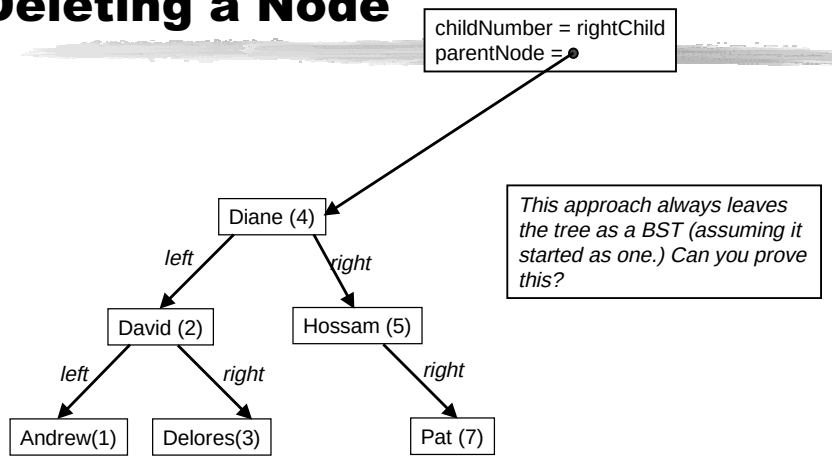
Deleting a Node



Deleting a Node



Deleting a Node



This approach always leaves the tree as a BST (assuming it started as one.) Can you prove this?

Adding a Node to a Tree

```
class TreeRef {
    ...
    public void deleteNode (TreeRef r) {
       StudentTree t = getNode();
       Util.assert (t != null,
                    "Attempt to delete null tree element");

       // To delete the node, replace it by its left subtree,
       // then add its right subtree
       StudentTree leftTree = t.getChild (StudentTree.leftChild);
       r.setNode (leftTree);  // step 2
       r.addTree (t.getChild (StudentTree.rightChild)); // step 3
    }
    ...
}
```

Note: doesn't keep tree balanced! Can you construct an example showing why not?

Implementing the Dictionary

- The dictionary interface is implemented by a class *TreeDictionary*
- This class contains a *TreeRef* representing the root of the tree containing the students
- Operations in dictionary are mapped to tree operations
  - *addStudent()* implemented by *addTree()*
  - *removeStudent()* implemented by *deleteNode()*
  - *…etc.*

```
public class TreeDictionary implements StudentDictionary {
    // The tree containing the students
    private TreeRef students =
              new TreeRef (new StudentTree(null),StudentTree.leftChild);

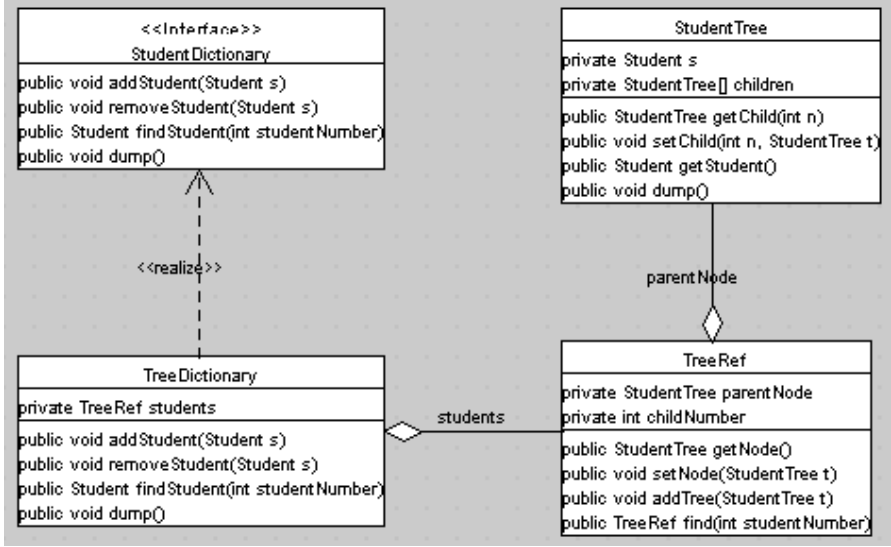
    public void addStudent (Student s) {
       // Create a tree node for this student and add it into the tree.
       StudentTree t = new StudentTree (s);
       students.addTree (t);
    }

    public void removeStudent (Student s) {
       students.deleteNode (students.find (s.getStudentNumber()));
    }

    public Student findStudent (int studentNumber) {
       // Look up the student with this student number
       TreeRef r = students.find (studentNumber);

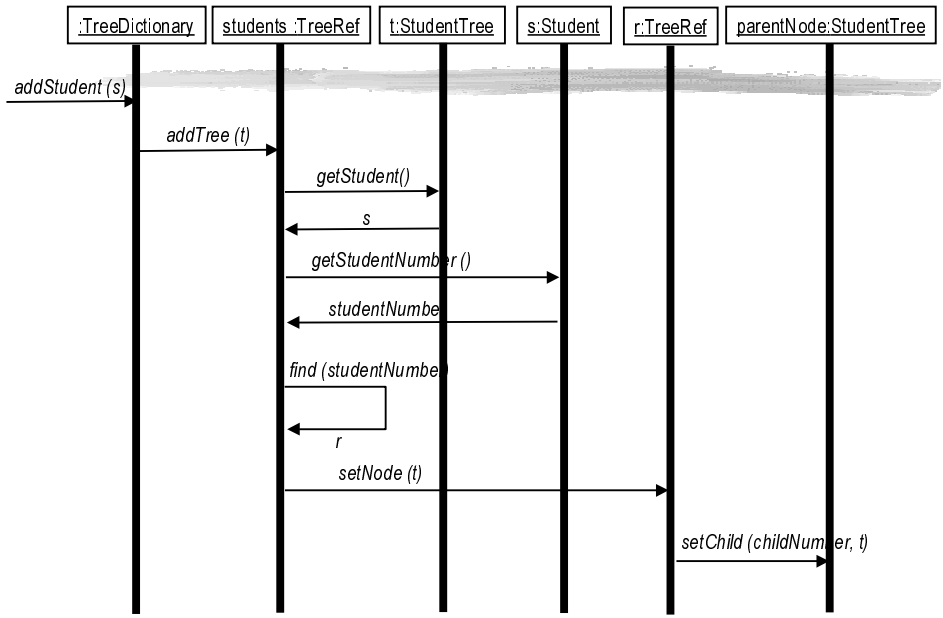
       // Get the actual tree node. If this is null, there is no
       // such student. Otherwise, return the student.
       StudentTree t = r.getNode();
       if (t == null) {
           return null;
       } else {
           return t.getStudent();
       }
    }
}
```

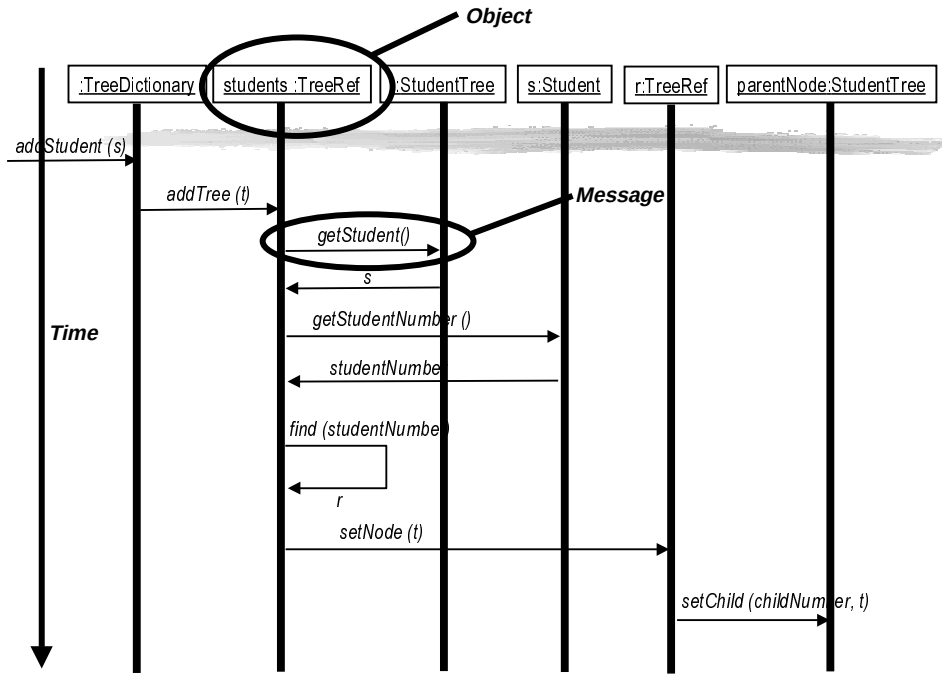
Recall structure of application...



Sequence Diagrams

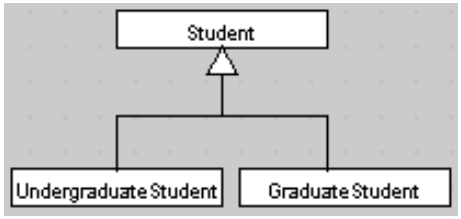
- Relationships between classes can be illustrated by sequence diagrams
  - show scenarios of how objects communicate
  - e.g., scenario of adding a student into the tree, scenario of finding a student



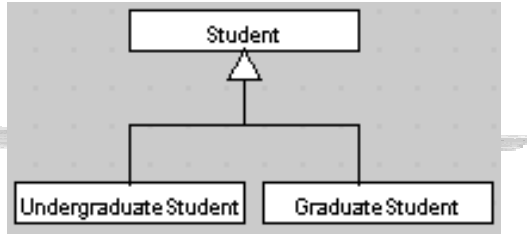


A note on typing...

- The dictionary is defined over objects of type *Student*
- But we want to use it for all kinds of students - including *UndergraduateStudent* and *GraduateStudent*



A note on typing...



- In general, can always assign up the class hierarchy

```
Student s = new GraduateStudent ("Fred", 123,
  "Comparative Analysis of Soap Operas")
```

- To assign down the class hierarchy, must explicitly *cast* the object to the desired type

```
GraduateStudent g = (GraduateStudent s);
```

  - Only works if *s* actually is a *GraduateStudent*

- Can test to see what class an object really belongs to

```
if (s instanceof GraduateStudent) {
   System.out.println ("Got a grad student!");
}
```

Summary

- Use of Tree class allows us to efficiently implement student dictionary
- Key to implementation is split of *StudentTree* and *TreeRef* types
- Single method *findStudent* used to find, add and delete nodes
- Complexity of algorithm contained in this method; other methods simple