- review traversal exercise
- implement binary search trees
- implement traversals
- full implementation

Next week and homework

- next week: finish implementation of binary trees; hashing; graphs
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, sections 17.3 on bst removal; 16.4

Lab

- final lab is assigned. See Canvas for due date

Review last week

- introduced trees, particularly binary search trees (bst)
  - the most useful case – used for fast binary search
- introduced tree traversals, pre-, in- and postorder
- looked at an example application, using binary trees to compress text

Introduction to this week

- review the traversal exercise from last week
- develop code to implement bsts and traversals
- develop a full generic implementation with an interface
- the last lab, using binary trees, is assigned this week

Review traversal exercise
Objective: review the traversal exercise assigned last week

Binary search tree exercise

- the exercise was, on a piece of paper:
  - build the binary search tree from the following input stream, then write the result of an inorder traversal

    2 3 8 4 6 9 1 5 7

- building the binary search tree from the arriving data gives:

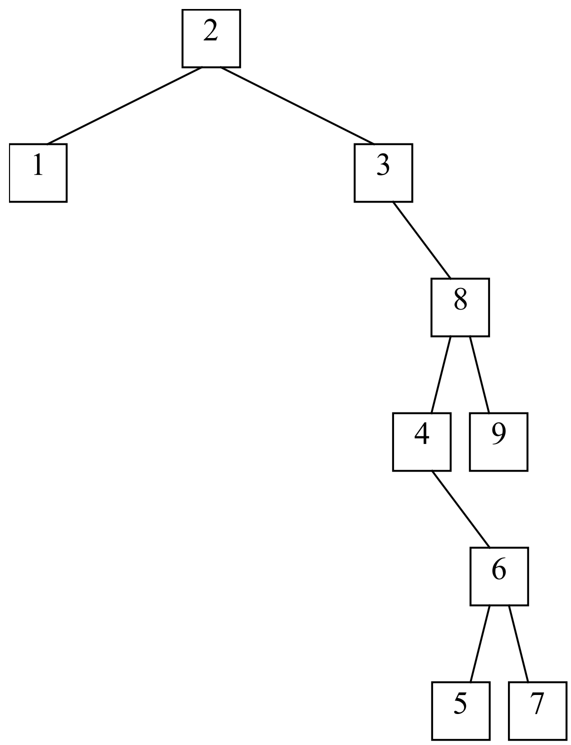

Figure 1 the binary search tree

  - (see that this data gives a tree that is not particularly well balanced)

- an inorder traversal is:

  left
  root
  right

  - this gives:

1 2 3 4 5 6 7 8 9

Summary

- see that an inorder traversal of a binary search tree gives data in sorted order!
  - exactly as the name implies
  - cool!
  - will see how the other traversals are used this week

Implement binary search trees
Objective: develop a binary search tree

Simplest possible implementation first
- keep it simple – no generic data types, interfaces
  - assume we'll build the bst from a stream of arriving data
  - assume no duplicates for now
  - will start from the simplest classes and methods and build upwards from these. This is developing 'bottom-up'

Node in a binary tree
- a node in a binary tree has the information, and two references:

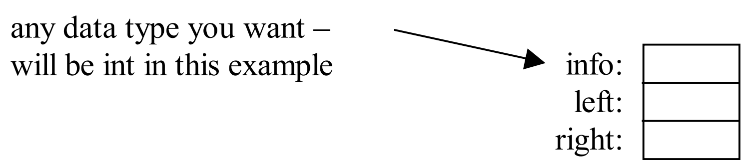

Figure 2 a node in a binary tree has two references

```
public class Node
{
    protected int info;
    protected Node left;
    protected Node right;
```

  - using protected as usual for our convenience

Binary search tree class
- a binary search tree has a root, which must be initialized when the tree is first created. Very straightforward:

```
public class BSTree
{
    private Node root;

    public BSTree()
    {
        root = null;
    }
```

<u>Inserting a new node into the bst</u>

- we build a bst from a stream of input data e.g.

20 10 18 25 23 16 15

root is a special case

Figure 3 build a bst from a stream of input data

  – build the tree from this arriving data. See that each time we add the new node as a child of an existing parent node:

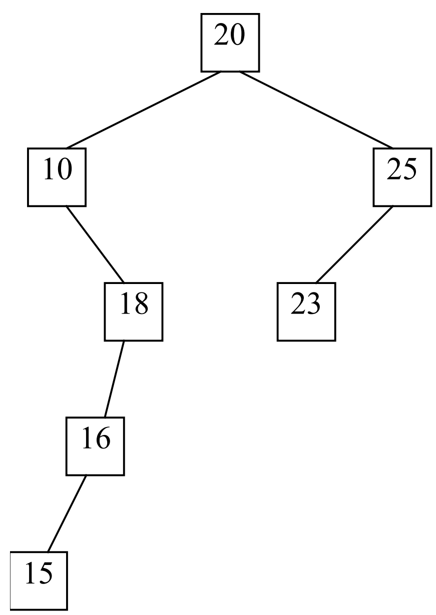

Figure 4 resulting binary search tree

  – notice that we never need to add a new node to a node that already has 2 children

  – so each time we need to find a reference to the parent node where our new child will be added

- developing this non-recursively, will use our old idea of lead and lag references to set the reference to the parent

  – remember, we used lead and lag to delete last element from a linked list e.g.

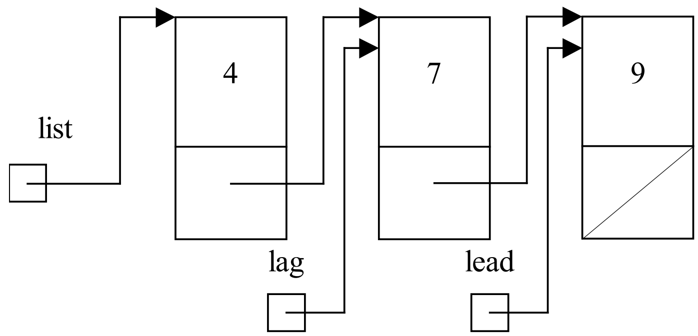

Figure 5 we used lead and lag to delete last element from a linked list

- here, needed to update the next-to-last element, requiring 2 references:

  `lead` – advanced first
  `lag` – follows behind

- will use the same idea for the bst

  `lead` is the lead reference – moved first
  `lag` is the lag reference – follows behind

  - start both at root

  - advance down the tree following left or right branches, until `lead` falls off, leaving `lag` pointing to the node where we will add the new node. Very cool!

- see this in Java:

```
public void insertBST(int x)
{
    if (root == null)
        root = new Node(x);
    else {
        Node lead, lag;
        lead = lag = root;
        while (lead != null) {
            lag = lead;
            if (x < lag.info)
                lead = lag.left;
            else
                lead = lag.right;
        }
        //lead fell off tree, lag is parent of new node
        if (x < lag.info)
            lag.left = new Node(x);
        else
            lag.right = new Node(x);
    }
```

```
    }
```

– nice!

Search the bst before inserting a node

- note that this method assumes that x is not already in the bst
  - because we assumed no duplicates, for simplicity
  - (it would be a disaster if a duplicate does occur, because the method would actually create another node with the same value in the tree, replacing any existing right child!)
  - (BTW, for safety, it would be easy here to test for an existing subtree, and either throw an exception or return a Boolean for fail, if a duplicate occurs)
  - or the easiest solution here is that we always have to search the bst first, to decide if a new node has to be inserted

Searching the bst

- the `find()` method should return a reference to the information if it is in the bst, or `null` if not found
  - this way the program can directly manipulate the information if it's present
  - implementation is straightforward e.g.:

```
public Node find(int x)
{
    Node r = root;
    while (r != null && r.info != x) {
        if (r.info < x)
            r = r.right;
        else
            r = r.left;
    }
    return r;
}
```

Summary

- from Canvas, 'Implementation of binary trees' module, Example programs, download, read, run and understand my `Simple binary search tree` example program

- see in `Tester::main()` that I hardcoded the arrival of the data to create the example bst given above

- next we'll see how to implement the different tree traversals, that visit the nodes in different orders

- (BTW, this example is just one way to implement bst)

Implement traversals
Objective: see an important use of recursion

Remember the three traversal orders

- introduced the three traversal orders last week:

  – preorder – "root left right"

  – inorder – "left root right"

  – postorder – "left right root"

- to implement, have to store info about which nodes to return to after reaching an edge of the tree. Several different possibilities:

  – stacks – explicitly push and pop addresses of nodes to return to

  – recursion – does the same thing, but the stack is implemented transparently for us, we don't have to code it

  – lists – thread nodes together using a list of nodes to return to

Recursive implementation is particularly simple and clear

- e.g. here's recursive preorder traversal:

  "root left right"

```
private void pretrav(Node r)
{
    if (r != null) {
        System.out.print(r.info + " ");      //root
        pretrav(r.left);                     //left subtree
        pretrav(r.right);                    //right subtree
    }
}
```

  – note that this is a private helper method, inside the `BSTree` class. Reason for this is that it must be called first with `root` as the param, and `root` is private to the `BSTree` class

  – so the public preorder traversal method is:

```
public void preorder()
{
    pretrav(root);
```

```
        System.out.println();
    }
```

– giving the call in `Tester::main()`:

```
t.preorder();
```

- the two other traversals are equally straightforward, just order the recursive calls appropriately

<u>Purpose of each traversal</u>

- here's our example bst again, built when we run my `Simple binary search tree` example program on this stream of input data:

20 10 18 25 23 16 15

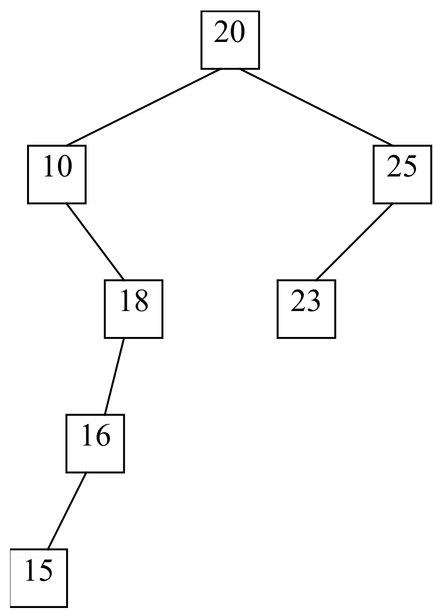

Figure 6 our example bst

– and here's the output from the program for each of the different traversals

preorder: 20 10 18 16 15 25 23

inorder: 10 15 16 18 20 23 25

postorder: 15 16 18 10 23 25 20

- we already know that, as the name implies, an inorder traversal gives the bst in sorted order. Very useful!
  - (so `toString()` for a bst could just be an inorder traversal)
  - the other traversals are also useful
- postorder traversal is useful in languages where we have to manually delete a binary tree
  - here we want to free each node only when all its subtrees have been freed i.e. visit the root last
  - see that the "left right root" postorder traversal does this exactly
- preorder traversal is used when copying a binary tree to a file
  - see that when we write the information to file using preorder traversal…
  - …then read it back from the file, we build the original binary tree
  - (BTW, notice however that none of the traversals preserve the original order of the information)

<u>Summary</u>

- we've seen that recursion gives a really simple, clear implementation of tree traversals

Full implementation
Objective: now make the implementation generic, and with an interface

Make my simple implementation generic

- now want to improve my simple implementation, so that we can have a bst of generic objects e.g.

```
BSTree<WordCount> t = new BSTree<WordCount>();
```

  – will need a bst of `WordCount`s for the next lab

  – will develop my final example program in three steps

1. First, make bst nodes 'comparable'

- an essential bst property is that we can 'compare' nodes – that a node must be less than, equal to or greater than another, to be placed in the bst

  – now that nodes can be of any data type, the way to ensure they are always comparable is by using the Java library `Comparable` interface

  – we do this by changing our `Node` class to contain a `Comparable` object

  – e.g. was:

```
public class Node
{
    protected int info;
    . . .
```

  – here `info` was declared as an `int`, to make the first example as simple as possible – just a bst of integers

  – now becomes:

```
public class Node
{
    protected Comparable info;
    . . .
```

  – (then made the other obvious changes in the class due to this change in data type)

  – now `info` can be of any data type, providing it implements the `Comparable` interface and is therefore guaranteed to be comparable

- so now the `Node` class contains a `Comparable` object, named `info`

- for example, strings are comparable, so we can now have a bst of strings e.g.

```
public static void main()
{
    BSTree t = new BSTree();

    //insert some new words into the tree
    t.insertBST("once");
    t.insertBST("upon");
    t.insertBST("a");
    t.insertBST("time");
    t.insertBST("in");
    t.insertBST("the");
    t.insertBST("west");

    //print in alpha order
    t.inorder();
    . . .
```

  - from Canvas, 'Implementation of binary trees' module, Example programs, download, read, run and understand my `Comparable` example program

  - see by running the program that the words are sorted, and we can search for words

  - let's look at the important points of this implementation

<u>Comparable objects have the `compareTo()` method</u>

- all classes that implement the `Comparable` interface must implement the `compareTo()` method. Check the API docs, and this method returns an `int` e.g. `a.compareTo(b)` returns:

  - < 0 if a < b

  - 0 if a == b

  - > 0 if a > b

- so now we use the generic `compareTo()` method wherever we compare the info value of nodes

  - works for all classes that implement the `Comparable` interface

  - e.g. `String`, `WordCount`, …

- e.g., in my example program we must re-write the `BSTree` class `find()` method:

```
     public Comparable find(Comparable obj)
     {
         Node findNode = new Node(obj);
         Node r = root;
//         while (r != null && r.info != x) {
         while (r != null && r.info.compareTo(findNode.info) != 0) {
             if (r.info.compareTo(findNode.info) < 0)
                 r = r.right;
             else
                 r = r.left;
         }
         if (r == null)
             return null;
         return r.info;
     }
```

   – here I've commented out the original comparison `r.info != x` which worked for `int`, but will not work correctly for `String`

   – instead, to compare any two `Comparable` objects, such as `String`:

```
r.info.compareTo(findNode.info)
```

   – see in the syntax that `r.info` is a reference to the `String` in the bst we're currently looking at, and `findNode.info` refers to the `String` we are trying to find

   – (note that in the example program I actually introduced a `comp` variable to avoid having to do the relatively expensive `compareTo()` method call more than once per loop iteration. May be more efficient, but is not so simple and clear. Your choice)

- see similarly in the `insertBST()` method, replaced the original comparison when `info` was an `int`, simply:

```
if (x < lag.info)
```

   – to now use the generic `compareTo()`, would be:

```
if (newNode.info.compareTo(lag.info) < 0)
```

   – (but again I use a variable `comp` in my example program, to avoid more than one expensive call to `compareTo()`)

- BTW, also notice that I changed `find()` to return a reference to the information, NOT to the bst node
  - this is more reasonable, because it's always the information stored in the data structure we're interested in, not the data structure or its implementation
  - with this reference, the program is able to access the information directly and change it if necessary
  - made the change in this example program because the information for the first time is an object, not an `int`
  - see the change in the code at the end, necessary to avoid a reference through a null reference, if the word is not found:

```
public Comparable find(Comparable obj)
{
     Node findNode = new Node(obj);
     Node r = root;
     int comp;
     while (r != null &&
       (comp = r.info.compareTo(findNode.info)) != 0) {
          . . .
     }
     if (r == null)
          //not found
          return null;
     return r.info;
}
```

- (also noticed that I dropped the traversals that we will not be using)

2. <u>Now make generic with a type parameter</u>

- this implementation works fine with `String`, is a good beginning, but it is not typesafe
  - information in the bst can be anything that implements `Comparable` i.e. `String`, `Integer`, …
  - so we could legally build a bst of `String` then try to insert an `Integer`
  - would compile, because `String` and `Integer` are both comparable
  - but would crash with a class type exception, because we cannot compare a `String` to an `Integer`

- we fix this problem by making the implementation properly generic with a type parameter, while still maintaining the comparable property
  - the type parameter ensures that everything in the tree and everything we try to do with it all have the same type, or are related appropriately by inheritance
  - and we can also constrain the type parameter so that it has to be comparable
  - perfect!
- my next example program demonstrates a generic bst, of `WordCount` objects, e.g.

```
public static void main()
{
    //create generic BST, of WordCount here
    BSTree<WordCount> t = new BSTree<WordCount>();

    t.insertBST(new WordCount("once"));
    t.insertBST(new WordCount("upon"));
    t.insertBST(new WordCount("a"));
    t.insertBST(new WordCount("time"));
    t.insertBST(new WordCount("in"));
    t.insertBST(new WordCount("the"));
    t.insertBST(new WordCount("west"));

    //print in alpha order
    t.inorder();
    . . .
```

  - from Canvas, ‘Implementation of binary trees’ module, Example programs, download, read, run and understand my `Generic` example program
  - see by running the program that the `WordCount` objects are sorted in alphabetical order, and we can search for `WordCount`s by words
  - let’s look at the important points of this implementation

Constrain the generic class type parameter to be comparable

- Java has a syntax that constrains a generic class type parameter in exactly the way we want
  - e.g., starting with the simplest bst class, for `Node`:

```
public class Node<E extends Comparable>
{
    …
}
```

  - this declares a generic class named `Node<E>` in the usual way, where the type of `E` will be set when an object of the class is created
  - (note that `E` is used by convention for an element type in a collection)
  - crucially, the `extends` syntax here constrains `E` that it must **extend or implement** the `Comparable` interface
  - exactly what we want!

- again, the name of this generic class is `Node<E>`, so the declarations for `left` and `right` become:

```
public class Node<E extends Comparable>
{
    protected E info;
    protected Node<E> left;
    protected Node<E> right;
```

  - so the class still contains a reference to an object named `info`, where the data type `E` has to implement the `Comparable` interface
  - `left` and `right` are still references to the left and right subtrees

- (BTW, generic type parameters are not yet fully implemented in Java because they are still so new. This can be confusing, as syntax that should not compile unfortunately does

  - e.g. in the example above, this compiles, even though it should not:

```
    protected Node left;
```

  - this is due to 'type erasure', where the JVM erases generic type parameters and replaces them with ordinary Java types
  - obviously, we must continue to write generic type parameters correctly, for when they will be implemented more stringently in future)

- see the similar changes when declaring the generic `BSTree<E>` class, where `E` is constrained to classes that are comparable:

```
public class BSTree<E extends Comparable>
{
    private Node<E> root;
```

  - then see the other changes required to use generic type parameter `E`, and bst node type `Node<E>`

The example class `WordCount` must be comparable

- `WordCount` is just an example class, which we will use in the next lab. Obviously, it must be comparable

  - the Java library interface `Comparable` is a generic class that takes a type param, so the declaration would be:

```
public class WordCount implements Comparable<WordCount>
{
    protected String word;
    protected int count;
    protected int list;
    . . .
```

  - (see that a `WordCount` object contains a word, a count of how many times the word occurs, and a placeholder for a list of where it appears. More on this later)

  - the declarations says that `WordCount` implements `Comparable`, so we are allowed to put `WordCount` objects into our generic bst

  - the `Comparable<WordCount>` part constrains the class, that `WordCount` objects can be compared only to other `WordCount`s, and not to more general objects

  - this is safer, because `WordCount`s are so specialized, is exactly what we want. The comparison is simply:

```
public int compareTo(WordCount other)
{
    return word.compareTo(other.word);
}
```

  - remember that `word` is a `String` here. `compareTo()` has been provided for `String`, decides which string occurs first in alphabetical order

- BTW, also notice that `toString()` has been provided for `WordCount`, for later

3. Add a bst interface

- finally, now want to improve this generic implementation by adding a bst interface

- adding an interface is not necessary, since it doesn't make sense to implement a bst in another way, by using arrays for example
- however, using an interface forces us to declare the essential primitive operators that all bst implementations must offer
- which makes the idea of bst more simple and clear

• e.g. here's our generic `BSTreeInterface<E>` interface

```
public interface BSTreeInterface<E extends Comparable>
{
    void insertBST(E obj);

    E find(E obj);

    public String toString();
}
```

- (BTW, BlueJ does not allow us to create a generic interface directly using the **New Class…** button. Instead we can create a regular interface, then manually edit in the type parameter)
- again the interface uses the `E` type parameter, for an element in a collection
- `E` again `extends` the `Comparable` interface, so that all objects must implement `compareTo()` to be comparable
- see how the interface clearly identifies the essential bst primitive operations of insert and find, that all implementations must provide
- also, that it makes clear a `toString()` method has to be implemented, since all Java programmers expect `toString()` for every class

• let's decide that `toString()` should traverse the bst inorder, to print out the `WordCount` information at each bst node

- so it's just our inorder recursive implementation
- except that `toString()` has to build and return an enormous `String` that contains all this information
- therefore uses `StringBuilder` to do this more efficiently
- the `toString()` method in `BSTree<E>`:

```
public String toString()
{
    StringBuilder sb = new StringBuilder();
    intrav(root, sb);
    return sb.toString();
}
```

– the recursive in order traversal method rewritten to use `StringBuilder`:

```
private void intrav(Node<E> r, StringBuilder sb)
{
    if (r != null) {
        intrav(r.left, sb);
        sb.append(r.info.toString() + "\n");
        intrav(r.right, sb);
    }
}
```

– and use the Java library `String` class static method `String.format()` in `WordCount::toString()`, to build a formatted string from all of the class instance variables, even though we have not used `count` and `list` in this example. Formatting syntax is the same as for `printf()`:

```
public String toString()
{
    return String.format("%-12s %3d %3d", word, count, list);
}
```

- now the `BSTree<E>` class should be changed to require it to implement this interface. Becomes:

```
public class BSTree<E extends Comparable> implements BSTreeInterface<E>
```

  – this says that elements `E` in the generic `BSTree` class must be comparable

  – and that the bst interface methods must be implemented, also with comparable elements `E`

  – nice!

- BTW, we still declare the bst in `Tester::main()` without regard to the interface:

```
BSTree<WordCount> t = new BSTree<WordCount>();
```

  – because we will only ever have one implementation of bst

  – so the effect of the interface here is only to document and enforce the methods provided

<u>Summary</u>

- from Canvas, 'Implementation of binary trees' module, Example programs, download, read, run and understand my `Full BST` example program

  - demonstrates a generic implementation of bst

  - where items in the bst must all be of the same type, and comparable

  - and the bst primitive operators defined by the interface must be implemented

- will use this example program in the next lab

Next week and homework

- next week: finish implementation of binary trees; hashing; intro to graphs
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, sections 17.3 on bst removal; 16.4

Lab

- final lab is assigned. See Canvas for due date