**OO Programming**

- Information Hiding – ability to maintain control over private variables
- Data Abstraction – (use of information hiding) Internal representation plus a set of procedure to access and manipulate the data
- Dynamic Binding – defers operation control to run-time — decision making based on state of an object (polymorphism)
- Inheritance – structure of ADTs within a hierarchy (single inheritance) or network (multiple inheritance)

---

**An Example of Using Inheritance**

- We have learned several types of trees, such as binary trees, binary search trees, AVL trees, and heaps.
- We have learned various operations to manipulate trees, such as, traversals, insertion, and deletion.

**Question:** Do we have to implement four separate binary tree classes?
Several Observations

- In/pre/post order traversal methods will be identical for binary trees, binary search tree, and AVL tree classes.
- Insertion is not clearly defined for binary trees in general, but is well-understood for binary search trees, AVL trees, and heaps, although the methods of insertion are different among these tree types.
- The same observation applies to the deletion operation.

If we implement 4 different tree classes, there will be significant redundancy in coding (consider 4 identical implementations of the preorder traversal).

However, there are four types of trees! What is the way out?

Inheritance

```cpp
template <class T>
struct Binary_node {T data; Binary_node* left,right;};

template <class T>
class Binary_tree {
    public:
        Binary_tree(); ~Binary_tree();
        void inorder();
        void preorder();
        void postorder();
        bool search(T& entry); // How ?
        T& find_max();
    protected:
        Binary_node* root;
        ... auxiliary functions omitted ...
};
```
template <class T>
class Search_tree : public Binary_tree {
public:
    Search_tree();  ~Search_tree();
    bool search (T& entry);
    void insert (T& entry);
    void remove (T& target);
    T& find_max ();
protected:
    ... auxiliary functions omitted ...
};

• Class Search_tree is said to be derived from class Binary_tree
  ➥ Binary_tree is called the base/parent class.
  ➥ Search_tree is called the derived/child class.
• Search_tree inherits all the resources of its base class.
  ➥ In/pre/post order traversals and the tree assignment operators are automatically available to Search_tree.
• Search_tree overrides the search() and find_max() methods of its base class.
• Search_tree also provides capabilities not provided by its base class, namely the insert() and remove() methods.
- By default, public members of the base class can only be used by member functions of the derived class (inaccessible to users of the derived class).
- The keyword `public` is used to make those functions available to users of the new class.
- Private members of a base class cannot be accessed by derived classes.
- **Protected** members of a base class can be accessed by derived classes, but not others.

---

```
template <class T>
class AVL_tree : public Search_tree {
public:
    AVL_tree(); ~AVL_tree();
    void insert (T& entry);
    void remove (T& target);
protected:
    ... auxiliary functions omitted ...
};
```

`AVL_tree` inherits all the resources of `Search_tree` and `Binary_tree`.

`AVL_tree` overrides the `insert()` and `remove()` methods of `Search_tree`. 
A Dilemma

Do we want to derive class Heap from Binary_tree?

Pro: At least conceptually, heaps are (complete) binary trees. They do have the same set of functionalities provided by the Binary_tree class.

Con: The array representation of heaps is so different from that of general binary trees that all inherited resources need to be overridden.

- There are no right or wrong answers to this problem. Just tradeoffs.
- For the purpose of discussion, let us derive Heap from Binary_tree.

Deriving the Heap Class

template <class T>
class Heap : public Binary_tree {
  public:
    Heap();  ~Heap();
    void inorder();
    void preorder();
    void postorder();
    bool search(T& entry);
    T& find_max();
    T& remove_max();
    void insert (T& new_entry);
  protected:
    ... auxiliary stuffs omitted here ...
};
Class Hierarchy

Binary_tree

Search_tree

Heap

AVL_tree

How Resources are Shared in The Hierarchy

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>BST</th>
<th>AVL</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>traversals</td>
<td>by itself</td>
<td>from BT</td>
<td>from BT</td>
<td>by itself</td>
</tr>
<tr>
<td>search</td>
<td>by itself</td>
<td>by itself</td>
<td>from BST</td>
<td>by itself</td>
</tr>
<tr>
<td>insert</td>
<td>NA</td>
<td>by itself</td>
<td>by itself</td>
<td>by itself</td>
</tr>
<tr>
<td>remove</td>
<td>NA</td>
<td>by itself</td>
<td>by itself</td>
<td>by itself</td>
</tr>
<tr>
<td>assignment</td>
<td>by itself</td>
<td>from BT</td>
<td>from BT</td>
<td>by itself</td>
</tr>
<tr>
<td>find_max</td>
<td>by itself</td>
<td>by itself</td>
<td>from BST</td>
<td>by itself</td>
</tr>
</tbody>
</table>
Polymorphism

To treat objects of a derived class as objects of the base class.

```cpp
class X {int a, b;};
class Y : public X {int c;};

main ()
{
    X x, *x_ptr=&x;
    Y y, *y_ptr=&y;
    x = y;    // we lose y.c
    y = x;    // illegal
    x_ptr = &y; // legal;
    y_ptr = &x; // illegal
    x_ptr = y_ptr; // legal
    y_ptr = x_ptr; // illegal
}
```

Discussion

- If class Y is derived from class X, an object of Y should be considered to be of type X.
  - Since Search_tree is a derived class of Binary_tree, a binary search tree is also a binary tree.

- However, when a child object is stored in a variable of the parent class, its extended capabilities/resources are lost.
  - Statement "x=y;" will not copy y.c to x.

- A pointer to a base class can point to an object of the derived class.
  - A Binary_tree* can point to a Search_tree.
  - In our example, x_ptr = &y is legal.
    - So is x_ptr = y_ptr.
A Second Example

main()
{
    Binary_tree<int> bt, *bt_ptr;
    Search_tree<int> bst, *bst_ptr;
    heap<int> hp, *hp_ptr;
    AVL_tree<int> avl, *avl_ptr;

    bt = bst; bt_ptr = &bst;
    bt = hp; bt_ptr = hp_ptr;
    bst = hp; bst_ptr = &hp;
    bst = avl; bst_ptr = avl_ptr;

    bst_ptr = bt_ptr;
    avl_ptr = bst_ptr;
    hp_ptr = avl_ptr;
}

Wait a Minute ...

- Assume that avl is a correct AVL tree.
- Let bst_ptr = &avl;
- **Question**: Is the result of bst_ptr->insert(30); a correct AVL tree ?
- Asked in another way: Which insert() are we calling ?
  - Search_tree::insert() ? (as suggested by the type of the pointer bst_ptr)
  - AVL_tree::insert() ? (as suggested by the type of the object bst_ptr points to)
- Solution ?
Virtual Functions

```
template<class T>
class Binary_tree {
    ...
    virtual void insert (T& x);
    virtual void remove (T& x);
    virtual bool find (T& x);
    void inorder ();
    ...
};

template<class T>
class Search_tree : public Binary_tree {
    ...
    void insert (T& x);
    void remove (T& x);
    bool find (T& x);
};

template<class T>
class AVL : public Search_tree {
    ...
    void insert (T& x);
    void remove (T& x);
    ...
};

main()
{
    Binary_tree<int> bt, *bt_ptr;
    Search_tree<int> bst, *bst_ptr;
    AVL<int> avl, *avl_ptr;
    ...
    construct trees bt, bst, hp, and avl ...
}
bt_ptr = &bt;
bt_ptr->insert(30);  // the insert() of BT
bt_ptr->find(30);    // the find() of BT
bt_ptr->inorder();   // the inorder() of BT

bt_ptr = &bst;
bt_ptr->insert(30);  // the insert() of BST
bt_ptr->find(30);    // the find() of BST
bt_ptr->inorder();   // the inorder() of BT

bt_ptr = &avl;
bt_ptr->insert(30);  // the insert() of AVL
bt_ptr->find(30);    // the find() of BST
bt_ptr->inorder();   // the inorder() of BT

An Exercise

class A {
public:
    int x;
    A () {x = 1;}
    void f () {x += 1;}
    void g () {x += 10;}
    virtual void h () {x += 100;}
};

class B : public A {
public:
    void g () {x += 100;}
    void h () {x += 1000;}
};
main ()
{
    A a, *a_ptr = &a;
    B b, *b_ptr = &b;

    a_ptr->f(); a_ptr->g(); a_ptr->h();
    cout << a_ptr->x;

    b_ptr->f(); b_ptr->g(); b_ptr->h();
    cout << b_ptr->x;

    a_ptr = &b;
    a_ptr->f(); a_ptr->g(); a_ptr->h();
    cout << a_ptr->x;
}

Abstract Classes

Let us consider class Binary_tree again.

- We cannot insert anything to a Binary_tree and to remove anything from the tree.
- It makes no sense to actually create a Binary_tree object.
- However, the existence of the Binary_tree class makes sense.
  1. It defines the common features of all types of binary trees.
  2. It provides a place to implement tree traversals, to be inherited by derived binary trees.
- The solution is to define Binary_tree as an abstract class.
template <class T>
class Binary_tree {
public:
    Binary_tree(); ~Binary_tree();
    void inorder();
    void preorder();
    void postorder();
    Binary_tree& operator= (Binary_tree&);

    virtual bool search (T& entry);
    virtual void insert (T& entry) = 0;
    virtual void remove (T& entry) = 0;
private:
    ... We don’t care about private part here ...
};

- A virtual function that ends with \(=0\) is called a **pure** virtual function, meaning that you are not to provide implementation for the function.

- A class that has at least one pure virtual function is called an **abstract class**.

- You cannot initiate any object of an abstract class.

  ※ “Binary_tree g;” is illegal with the new `Binary_tree` definition.

  ※ “Binary_tree *g = new Binary_tree;” is also illegal.

- However, “Binary_tree *g = new AVL_tree” is legal, because an abstract class pointer can point to objects of its derived classes.