This lecture is about trees, which are another common data structure.

We'll be looking at binary trees, how they're represented and built.

We'll also look at ordered binary trees, which are a good way to store items in some order, such as alphabetical order.

Trees are also important in parsing (breaking a sentence into its parts), which we'll study later.

This type declaration is for a binary tree of strings. It's quite similar to the declaration for a list.

The tree comprises a number of nodes. Each node has two children, or sub-trees, which are themselves trees. A tree can be empty.

Here's an example.

```
#include <iostream.h>
#include <string.h>

typedef string Item;

class TreeNode {
    public:
        Item name;
        TreeNode* left;
        TreeNode* right;
};

typedef TreeNode* TreePtr;
```

Here are some access procedures for trees.

The empty tree is represented by the NULL pointer.

The access procedures that extract the parts of a tree are trivial.
This particular binary tree has an interesting property. It is an ordered binary tree.

For any node N, every node in the left sub-tree of N is alphabetically before N, and every node in the right sub-tree is alphabetically after (or equal to) N.

Ordered binary trees are very useful if we need to maintain an ordered sequence of items.

- It’s easy to insert a new element.
- It’s easy to traverse the tree in order.

Ordered trees are much faster for lookup than lists.

For any node N, every node in the left sub-tree of N is alphabetically before N, and every node in the right sub-tree is alphabetically after (or equal to) N.

Ordered binary trees are very useful if we need to maintain an ordered sequence of items.

- It’s easy to insert a new element.
- It’s easy to traverse the tree in order.

Ordered trees are much faster for lookup than lists.

- Suppose we want to insert “Katie” into this tree.
- “Katie” is after “John”, so it gets inserted into the right sub-tree of the root node.
- “Katie” is before “Mary” so it gets inserted into the left sub-tree of this node.
- “Katie” is before “Lucy” so it gets inserted into the left sub-tree of this node.

The left sub-tree of this node is empty, so we’ve reached the base case.

A new sub-tree is created comprising just “Katie”, and this becomes the new left sub-tree of “Lucy”.

Here’s a recursive procedure for inserting a new name into the tree, while preserving its ordered property.

```c
void insertName (string newName, TreePtr tree) {
    TreePtr newNode;
    if (tree == NULL) { // base case - empty
        newNode = new TreeNode;
        newNode->name = newName;
        newNode->left = NULL;
        newNode->right = NULL;
        tree = newNode;
    } else if (newName < tree->name) {
        insertName (newName, tree->left);
    } else {
        insertName (newName, tree->right);
    }
}
```
To print out all the names in the tree in alphabetical order, we just have to visit the nodes in the right order.

Here's a recursive procedure that does it. We visit the nodes in the sequence: left sub-tree, node, right sub-tree. (This is called **in-order traversal**.)

```cpp
void printNames (TreePtr tree) {
    if (tree != NULL) {
        printNames(leftChild(tree));
        cout << nodeName(tree) << endl;
        printNames(rightChild(tree));
    }
}
```
Example

```c
void printNames (TreePtr tree) {
    if (tree!=NULL) {
        printNames(leftChild(tree));
        cout << nodeName(tree) << endl;
        printNames(rightChild(tree));
    }
}
```

Example

```c
void printNames (TreePtr tree) {
    if (tree!=NULL) {
        printNames(leftChild(tree));
        cout << nodeName(tree) << endl;
        printNames(rightChild(tree));
    }
}
```
void printNames (TreePtr tree) {
    if (tree != NULL) {
        cout << nodeName(tree) << endl;
        printNames(leftChild(tree));
        printNames(rightChild(tree));
    }
}
Lecture 9 – Binary Trees

Example

```c
void printNames (TreePtr tree) {
    if (tree != NULL) {
        printNames(leftChild(tree));
        cout << nodeName(tree) << endl;
        printNames(rightChild(tree));
    }
}
```

Traversing the Tree II

- By swapping these two lines, the same procedure prints out all the names in reverse.
- You can also do **pre-order traversal** (node, left sub-tree, right sub-tree) and **post-order traversal** (left, right, node). But these don’t do anything useful here.

```c
void printNamesBackwards (TreePtr tree) {
    if (tree != NULL) {
        printNamesBackwards(rightChild(tree));
        cout << nodeName(tree) << endl;
        printNamesBackwards(leftChild(tree));
    }
}
```

Deletion Example I

- Suppose we want to delete “Bill” from this ordered binary tree.
- First we have to find “Bill”. So we search down the tree in the usual fashion until we've got a sub-tree that has “Bill” at the root.
- Our job now is to delete the root from this sub-tree.
Deletion Example II

• To delete the root of a sub-tree, we
  • replace it by the immediate (alphabetic) successor of the element it contains, and
  • delete the node where that successor came from.

• In general, the immediate successor of the element in the root node is the leftmost node in its right sub-tree. (Think about it.)

• In this case, the immediate successor of “Bill” is “Fred”.

• So we replace “Bill” by “Fred”.

• Then delete the node where “Fred” came from.

Deletion Example III

• Here’s the final tree.
• Note that we have preserved its orderedness.

Deletion I

• Now we’ll look at the required C++ code.
• The code is quite tricky, and requires several mutually recursive procedures.

• First we have to find the node to be deleted.

• We check that the tree is not empty. If it is, the node to be deleted doesn’t exist, so we leave the tree as it is.

• We’ve found the node to be deleted if it is the root of the tree. We then call another procedure to remove the root.

• Otherwise, we carry on looking down the appropriate branch.

Deletion II

• Here’s part of the code for the removing the root.

• If the root node has no right sub-tree, then all we have to do is replace the whole tree by the root’s left sub-tree.

• Otherwise, we replace the root with its immediate successor in the tree, and remove the node where that successor was.

• The successor of a node is the leftmost node of its right sub-tree. The variable Leftmost takes on this value.

```
void deleteNode(string name, TreePtr &tree){
  if (tree == NULL) {
    if (nodeName[tree] == name) found
      deleteRoot(tree);
    else if (name < nodeName[tree])
      deleteNode(name, leftChild(tree));
    else
dataNode(name, rightChild(tree));
  }
}
```
• Finally we have a procedure that finds and deletes the leftmost descendant of a given node.
• First it has to find this node.
• If the node has no left sub-tree then it has no left descendants, and we've found it.
• So we call `DeleteRoot` again to remove the node, and we return the name it contained in the variable `Leftmost`.
• Otherwise, we continue moving down the tree, keeping to the left.

```cpp
void deleteLeftmost(TreeNode *tree, string &leftmost) {
    if (tree->left == NULL) {
        leftmost = tree->name;
        deleteRoot(tree);
    } else
        deleteLeftmost(tree->left, leftmost);
}
```

• The procedures for deleting a node from a binary ordered tree are *mutually recursive*.
• This means that procedure A calls procedure B, which calls procedure A again.
• In C++, we can't use a function before we've declared it, so when we write mutually recursive procedures, we have to make a *function prototype*.
• So, before the function `DeleteRoot()`, we put in a function prototype for the function `DeleteLeftmost()`. The actual code for `DeleteLeftmost()` comes later, after the `DeleteRoot()` procedure.

• Insertion and lookup in an ordered binary tree are, in general, more efficient than insertion and lookup in an ordered list.
• Intuitively, we can see why.
• To find an element in an ordered binary tree, the worst we ever have to do is search down to the lowest layer of the tree. If the tree has 4 layers, it can store \(1+2+4+8 = 15\) elements, but it only takes a maximum of 5 iterations of a loop to find any element.
• Contrast this with an ordered list of 15 elements. There it could take as many as 15 iterations around a loop to find an element.

• In a *balanced* ordered binary tree, each iteration of the lookup loop halves the number of elements left to search through.
• So on average, we can expect lookup to take roughly \(\log(N)\) steps.
• But the tree has to be reasonably well balanced to get good results. A completely balanced tree is one in which every node in every layer above the bottom layer has two children.
• When a tree is very unbalanced it is just like a list. So in the worst case lookup can take as long as lookup in a list.
• We can improve our insertion procedure by rebalancing the tree after each insertion. (We won't give details here.)