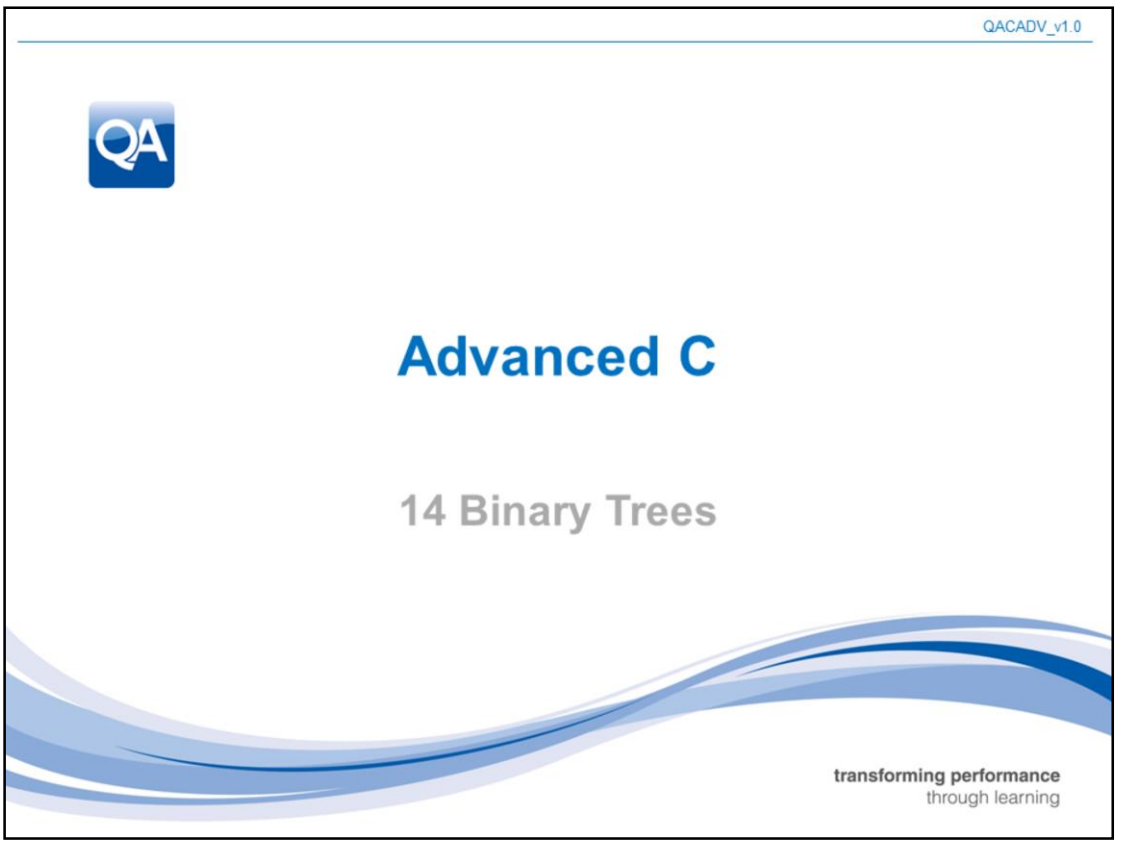
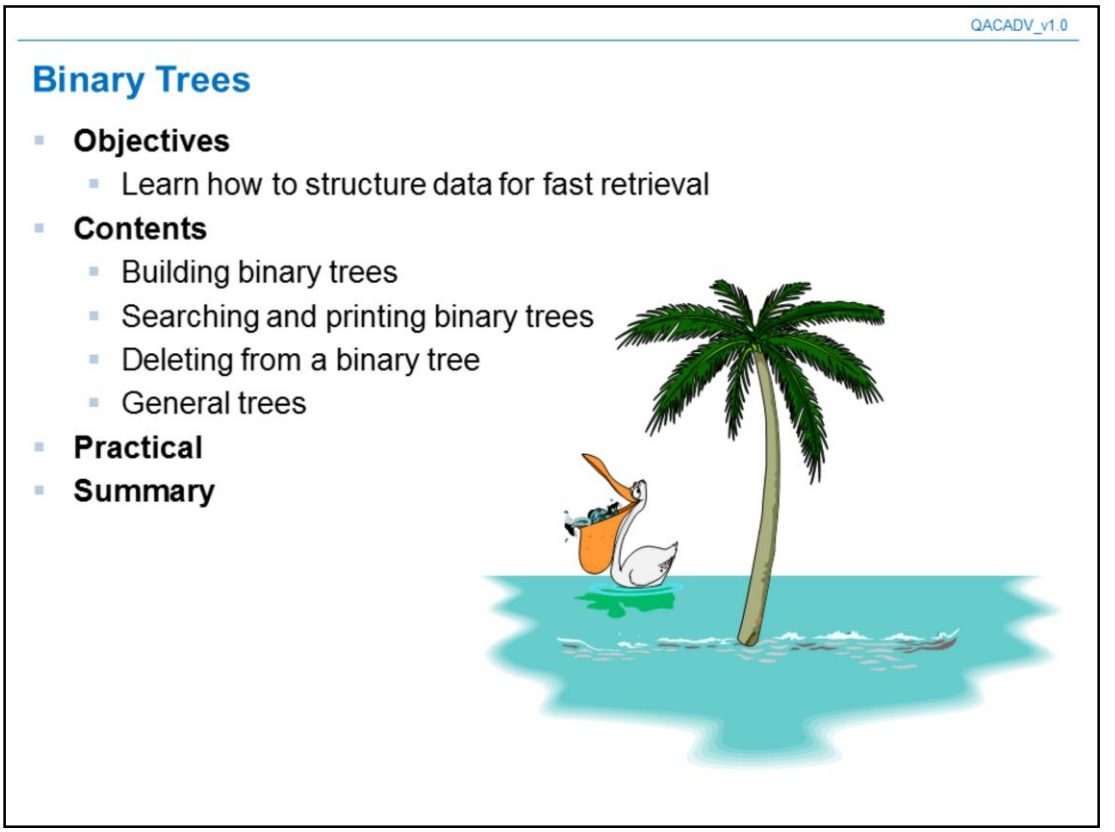

QACADV_v1.0
Advanced C
14 Binary Trees
transforming performance
through learning



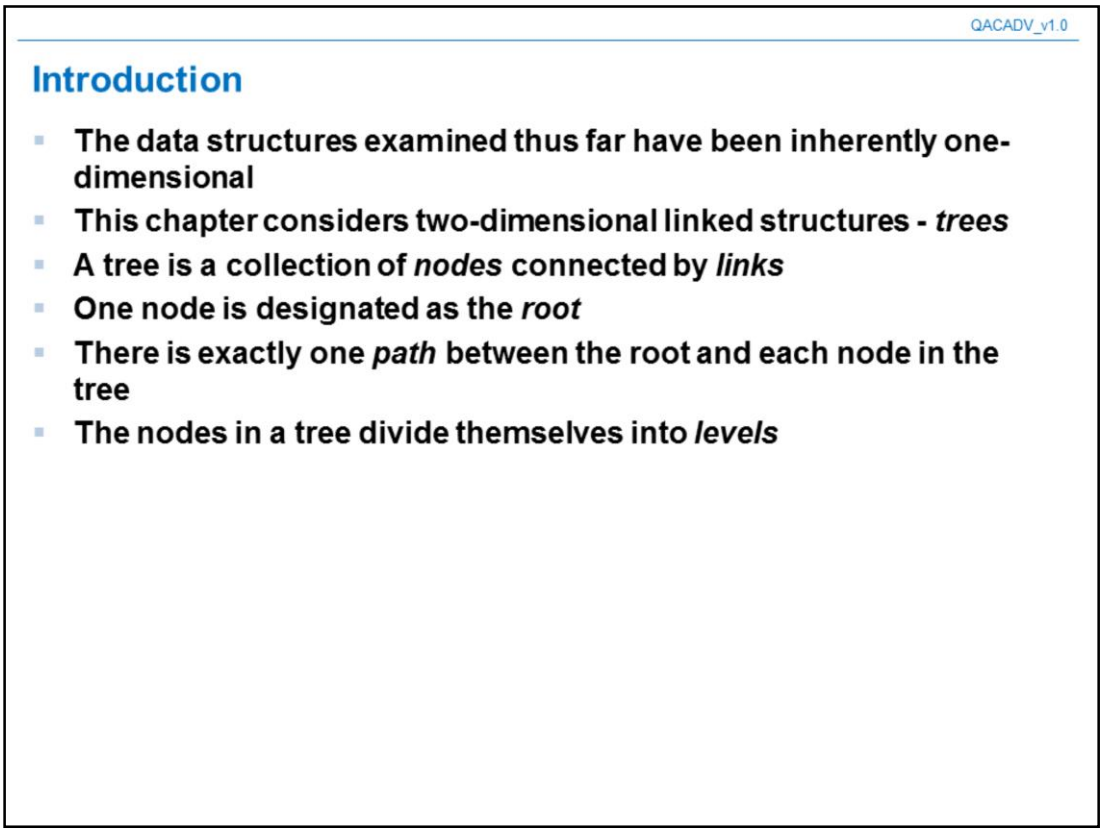
In the previous chapter, we looked at linked lists and saw how they may be used to store varying numbers of records during a program's execution.

Linked lists rely on dynamic memory allocation to ensure memory is used efficiently. Linked lists are ideal for unsorted data requiring sequential access, but are less appropriate if the data needs to be held in some sorted order. Imagine the situation where you need to find a particular data item in a linked list; the only way to navigate the list is to examine each node in turn, starting at the beginning of the list and continuing until the appropriate item is located (or the end of the list is reached).

Trees are more appropriate than linked lists when dealing with sorted data. A tree is a dynamic data construction in which items are held in strict sorted order. The search time is much better than for a linked list because a *binary chopping* technique is used when traversing a tree.

The most commonly used tree is a *binary tree*, where each node in the tree has at most two branches emerging from it. We shall concentrate on binary trees in this chapter. It is also possible to have more general trees, where each node can have any number of offspring. We shall look at this technique briefly as the chapter unfolds.

An appendix is dedicated to the discussion of AVL trees. These are more complex than binary trees and are primarily used to retain some balance in a tree, regardless of the order in which items are added. We shall discuss the theory of AVL trees and highlight the operations needed to maintain a tree's balance. Full source code for AVL tree-handling is provided on your systems.



The data structures that we have looked at so far during the course, such as linked lists, are inherently one-dimensional data structures. This chapter considers the concept of a *tree*, a two-dimensional linked structure that lies at the heart of some of the most important algorithms in existence today.

Trees are encountered in everyday life, and you are probably already familiar with the basic idea. For example, some people keep a record of their descendants in a family tree. This would be a rather complicated structure to represent within a program since each *node* (that is, a husband-and-wife pair) would have an arbitrary number of children (within reason, of course!).

The trees we will examine initially will have a *fixed* number of children per node. Later in the chapter we shall look at trees that have an *arbitrary* number of children per node.

The terminology used with trees follows that of a family tree; we speak of *parent* nodes, *child* nodes, *grandchildren*, *great grandchildren* and so on. One node in a tree is designated as the *root*. This is the ancestor node above which there are no other ancestors.

Only one *path* may be traced between the root and any node in the tree. Considering family trees again, this is fairly obvious. Some other data structures, such as *graphs*, have an arbitrary numbers of links between different nodes. Graphs are not discussed in this course.

Nodes are said to appear on different *levels* in the tree. For instance, a person would appear *two* levels below his or her grandparents in a family tree.

QACADV_v1.0

Representing nodes

- First, we will consider binary trees
- The same data structure is required as for a double-linked list:

```
typedef struct binary_node
{
    int                 data;
    struct binary_node * left;
    struct binary_node * right;
} bnode;
```

- The data member could be of any type

Double-linked lists and binary trees are closely related. Instead of using the member names `next` and `previous` we use `right` and `left`. The `data` item could have any type, depending on nature of the data you wish to hold in the tree. The following function creates a `bnode` structure using dynamic memory:

```
bnode * create_node(int val)
{
   bnode * p;
   p = (bnode *)calloc(1, sizeof(BNODE));
   if (p == NULL)
   {
      fprintf(stderr, "Insufficient memory\n");
      exit(EXIT_FAILURE);
   }
   p->data = val;
   return p;
}
```

Using `calloc` guarantees that the `right` and `left` pointers are `NULL`. This is important because newly-created nodes are always added at "leaf" positions in the tree, and therefore have no offspring initially.

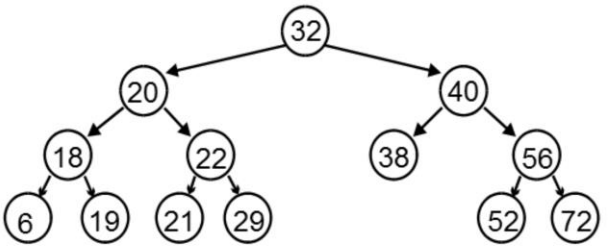
The following function destroys a node:

```
void destroy_node(bnode * p)
{
   free(p);
}
```

QACADV_v1.0

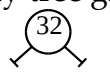
Building a binary tree

- **The values 32, 40, 20, 22, 56, 18, 6, 38, 52, 19, 72, 29 and 21 create the following tree:**

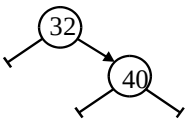


- **Tree is nicely balanced. For instance, only four comparisons are required to discover 98 is not in the tree**

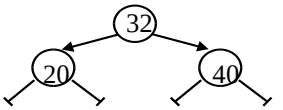
If the values `32`, `40`, `20`, `22`, `56`, `18`, `6`, `38`, `52`, `19`, `72`, `29` and `21` shown above are placed in the tree in the order listed, the tree will appear as shown in the slide above. Adding `32` to an empty tree gives the following:



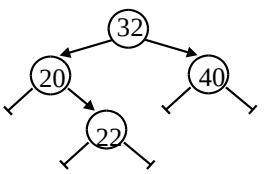
The tree contains one node with no children. The value 40 must be added to the *right* of this node since its value is greater than the value 32 stored in the root node:



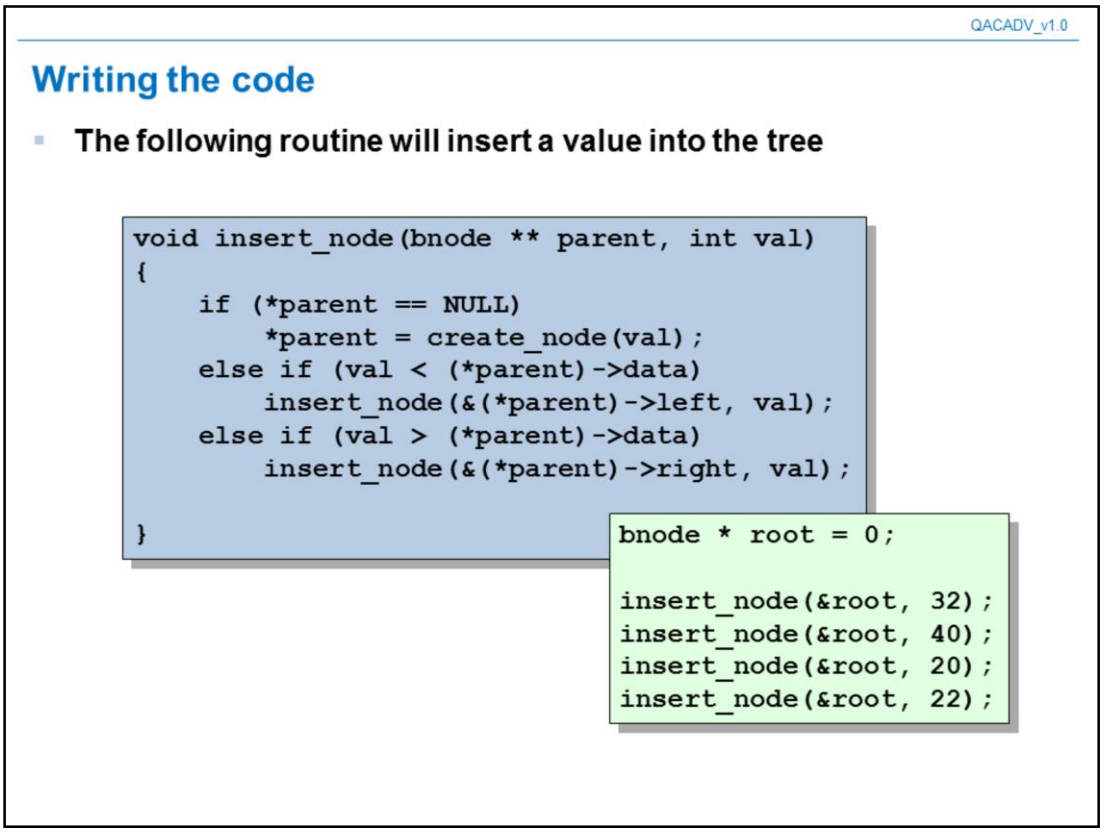
20 must be placed to the *left* of the root node, since 20 is less than 32:



22 is added next. Since this is less than 32, we move *left*; since it is greater than 20, we then move *right*:



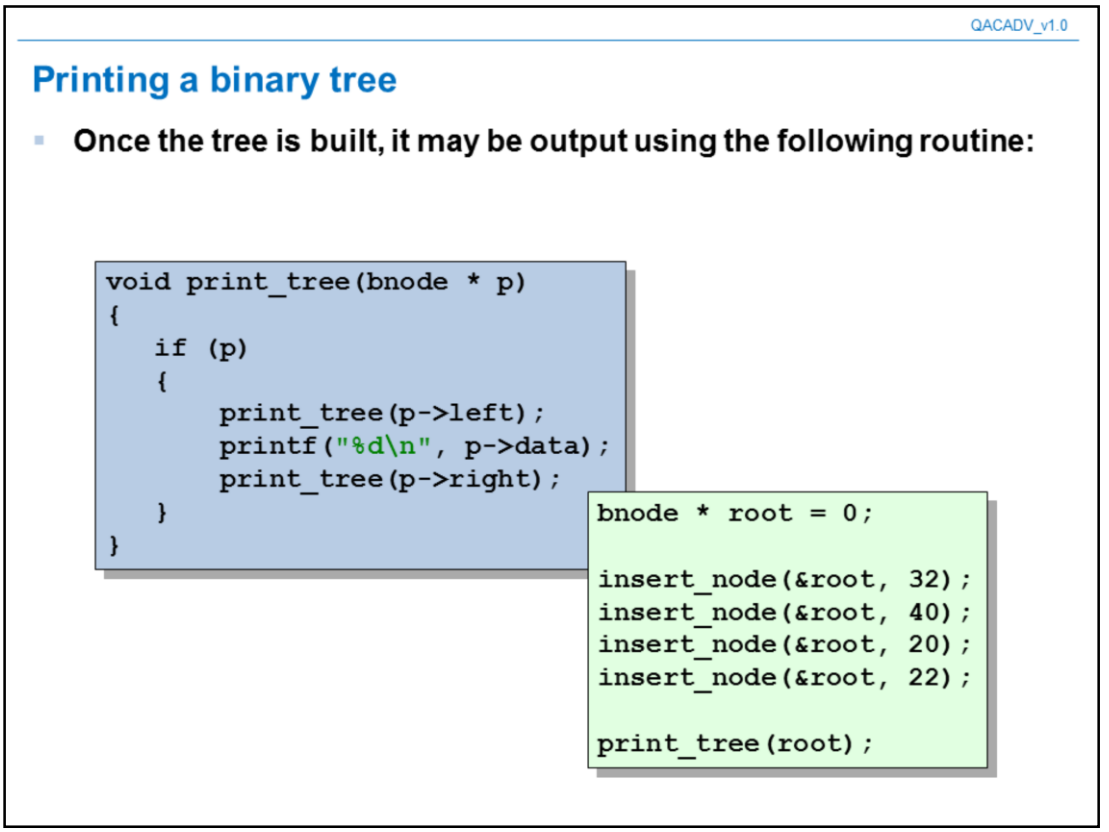
The tree shown in the above slide is built up by this process; the resulting tree is quite nicely balanced. For the moment we will define "balanced" as an aesthetic quality. Later we will propose a more quantitative measure of "balance".



The recursive function `insert_node()` shown above adds a new value to the tree. When `insert_node()` is called, the *address* of the `root` variable is passed to the function. This allows the function to alter the root variable. The first time `insert_node()` is called, `root` is initialised to point at the first node created.

`insert_node()` can be implemented using iteration rather than recursion, but the code is somewhat more complicated and less elegant:

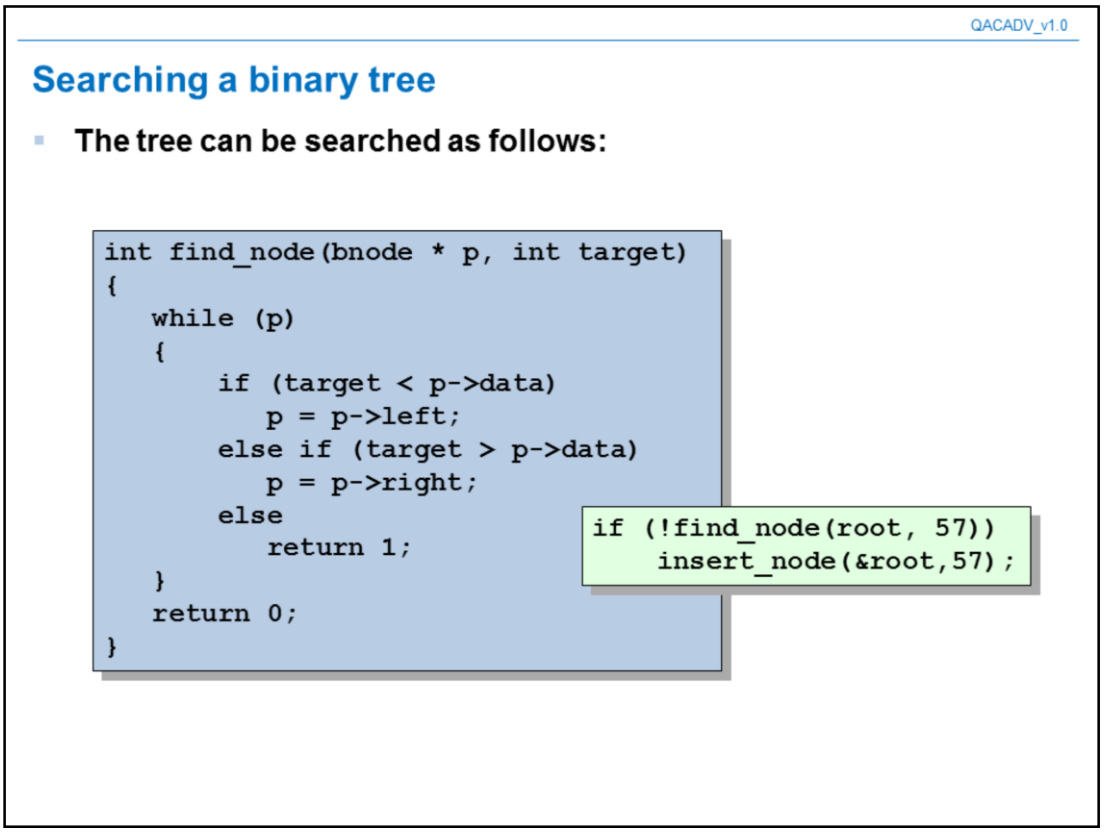
```
void insert_node(bnode ** root,  int val)
{
   bnode * p = *root;
   if (*root == NULL)
   {
      *root = create_node(val);   return;
   }

   for (;;)
      if (val < p->data)
      {
         if (p->left == NULL)
         {
            p->left = create_node(val);   return;
         }
         p = p->left;
      }
      else if (val > p->data)
      {
         if (p->right == NULL)
         {
            p->right = create_node(val);  return;
         }
         p = p->right;
      }
      else
         return;
}
```



The `print_tree()` function uses recursive techniques to print the whole of the tree. An iterative version of the function is shown below for completeness:

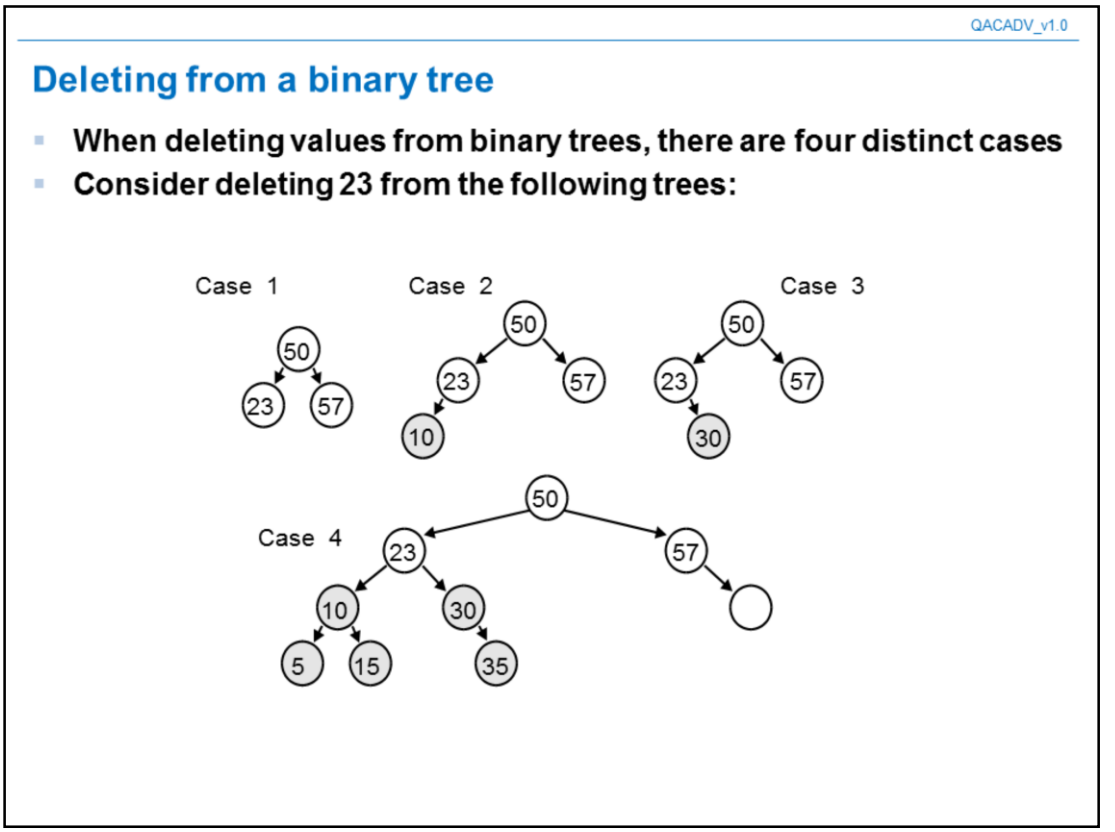
```
void iter_print_tree(bnode * p)
{
   for (;;)
   {
      if (p != NULL)
      {
         push(p);
         p = p->left;
         continue;
      }

      if (stack_empty())
         return;

      p = pop();
      printf("%d\n", p->data);
      p = p->right;
   }
}
```

Although it seems hard to believe, `iter_print_tree()` really does do the same as the recursive `print_tree()` function shown in the slide, which is an example of *recursion removal*[†]. The `push()`, `pop()` and `stack_empty()` functions do of course need to be implemented. This is a relatively simple task.



The function shown above searches for an integer value within the tree. A recursive version of this function is shown below:

```
int recursive_find_node(bnode * p, int target)
{
   if (p == NULL)
      return 0;

   if (target < p->data)
      return recursive_find_node(p->left, target);

   if (target > p->data)
      return recursive_find_node(p->right, target);

   return 1; /* p->data == target */
}
```



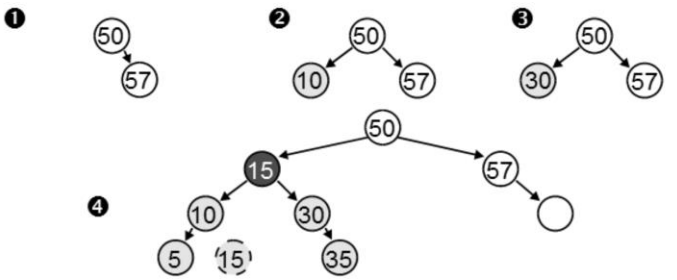
Many textbooks discuss how to *add* a value to a binary tree, but few books discuss how to *delete* a value from a tree. This is a pity because deletion is relatively straightforward. There are four cases that need to be considered:

1. The node to be deleted appears at the end of a branch and has no subtrees.
2. The node has one subtree to the left.
3. The node has one subtree to the right.
4. The node has two subtrees, one on the left and one on the right.

QACADV_v1.0

The four deletion cases

- If a node has no children, just delete it
- If a node has one subtree, replace the node with that at the root of the subtree
- If a node has two subtrees, replace the node with the *largest* value from the *left* subtree



- The code to do this is provided below (in the notes)

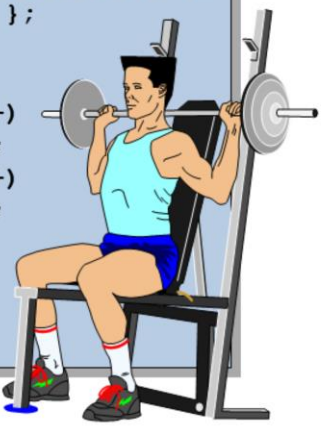
```
void remove_node(bnode ** root,  int val)
{
   bnode * d = *root,  ** pParent = root;
   static int static_local_destroy = TRUE;

   while (d != NULL && d->data != val)       /* point d at node to delete */
      if (val < d->data)
      {  pParent = &d->left;  d = d->left;  }/* point pParent at the field*/
      else                                   /* which points to d         */
      {  pParent = &d->right; d = d->right; }

   if (d == NULL)   return;                  /* Value NOT in the tree?    */

   if (d->right==NULL && d->left==NULL)      /* CASE 1; d has no children */
   {
      *pParent = NULL;
      if (static_local_destroy)  destroy_node(d);
   }
   else if (d->right==NULL || d->left==NULL) /* CASE 2/3, d has one child */
   {
      *pParent = (d->right == NULL) ? d->left : d->right;
      if (static_local_destroy)  destroy_node(d);
   }
   else                                      /* CASE 4: d has two children*/
   {
      bnode * gtChild = d->left;
      while (gtChild->right != NULL)         /* find the biggest value in */
         gtChild = gtChild->right;           /* the left subtree          */

      static_local_destroy = FALSE;          /* DON'T destroy biggest child*/
      remove_node(&d->left, gtChild->data); /* but "unsew" it.            */
      static_local_destroy = TRUE;           /* destroy the node next time */

      (*pParent)        = gtChild;           /* make the biggest child node*/
      (*pParent)->left  = d->left;           /* point to the children of   */
      (*pParent)->right = d->right;          /* the deleted node           */

      destroy_node(d);                       /* now destroy the node       */
   }
}
```

QACADV_v1.0

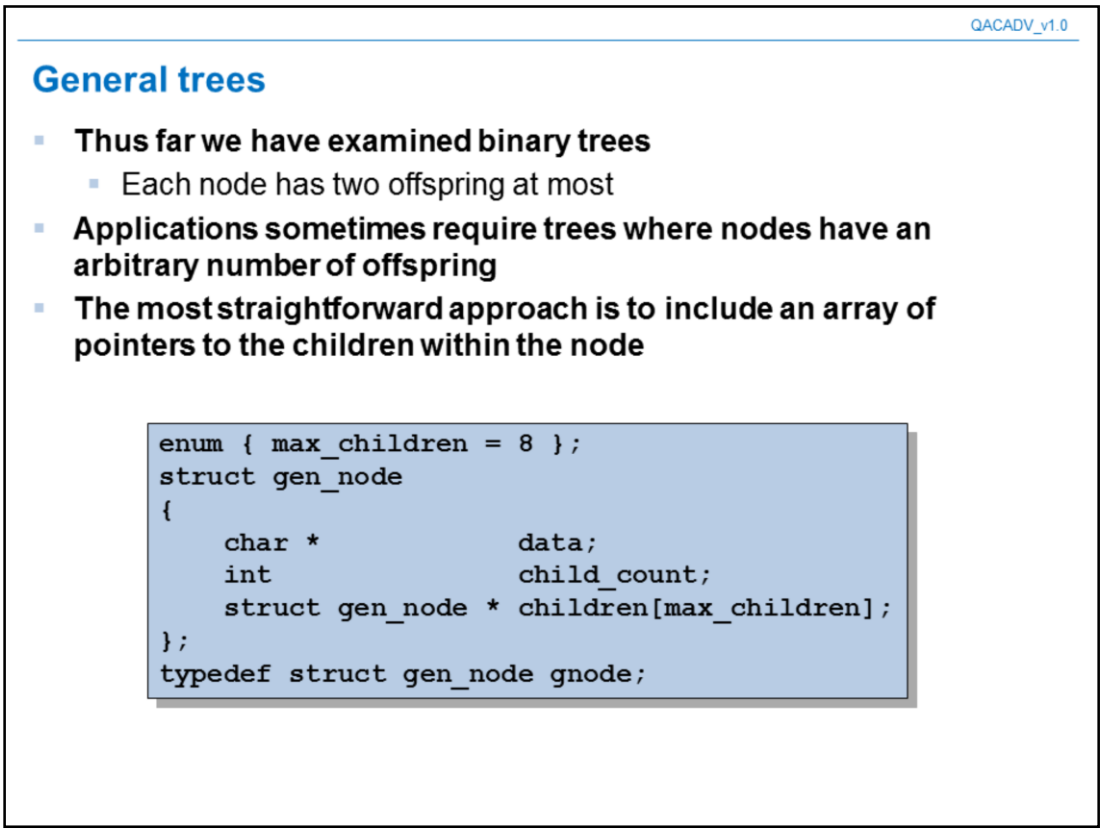
Exercise

- **What tree is produced by the following code?**

```
int main(void)
{
    size_t i;
    int insert[] = { 37,24,39,83,82,36,3,10,1,38 };
    int remove[] = { 10,3,39,37,24,36 };
    bnode * root = 0;

    for (i = 0; i < ASIZE(insert); i++)
        insert_node(&root, insert[i]);
    for (i = 0; i < ASIZE(remove); i++)
        remove_node(&root, remove[i]);
    print_tree(root);

    return 0;
}
```



Assuming that the `insert_node()` and `remove_node()` functions are available, and the `ASIZE` macro from the *Arrays* chapter is also provided, what will be printed by the program shown above?



Binary trees are excellent data structures for searching and sorting, but they can be a little cumbersome in some situations. For example, consider building a tree from the file system on your disk. In such a structure, a node (representing a directory) would need to point to an arbitrary number of child nodes (representing the files within the directory).

It would, of course, be possible to declare a number of *different* structures with fixed numbers of offspring, as follows:

```
struct node3
{
   int    how_many;
   void * left;
   void * middle;
   void * right;
};
```

```
struct node4
{
   int    how_many;
   void * left;
   void * mid_left;
   void * mid_right;
   void * right;
};
```

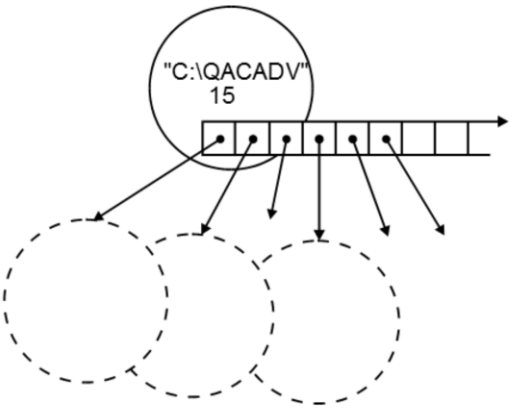
However, this approach introduces a number of problems. Most importantly, there is no way of telling what each of the pointers points to, i.e.. is the pointer pointing to a node with no offspring, a node2 with two offspring, or a node3 with three offspring?

Placing an array of pointers within the node is a better solution, since this allows us to use a single structure type to represent any number of nodes, up to the maximum array size. This technique is discussed on the following page.

QACADV_v1.0

Problems

- **If we represent a node in this way:**
  - What about nodes with no children?
  - What about nodes requiring more than *max_children* offspring?



The example shown above overcomes the difficulties discussed on the previous page, but unfortunately raises a number of new issues. For example, what if a node has no children at all? The node will still contain a *completely empty* array of `max_children` pointers.

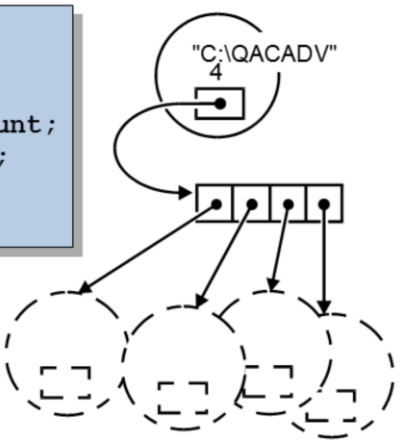
More seriously, what if one of the directories on the system has many hundreds of files within it? No matter what value we choose for `max_children`, the fact remains that there will be some upper limit that will, sooner or later, prove to be inadequate. Furthermore, using a large value for `max_children` to deal with the "nightmare scenario" results in a great deal of wasted space in the majority of situations.

QACADV_v1.0

General trees and dynamic memory

- **A better solution is to maintain a pointer to a block of dynamically allocated memory in which the addresses of the children are maintained**

```
struct gen_node
{
    char *              data;
    int                 child_count;
    struct gen_node ** children;
};
typedef struct gen_node gnode;
```



A better approach is to make use of dynamic memory. Each node can allocate just enough memory to suit its purposes. For example, a node with no offspring will not allocate an array at all, and therefore only suffers the overhead of a single (`NULL`) pointer. On the other hand, a node requiring sixty children will maintain a pointer to a block of memory holding sixty pointers.

What happens if the user creates a new sub-directory? The number of child nodes will have to be increased. We can use `realloc()` to increase the size of the pointer array to the exact size required; we no longer have to worry about creating an artificially large number of children (`max_children`) "just in case". The following function might be used to create a node in the "general" tree:

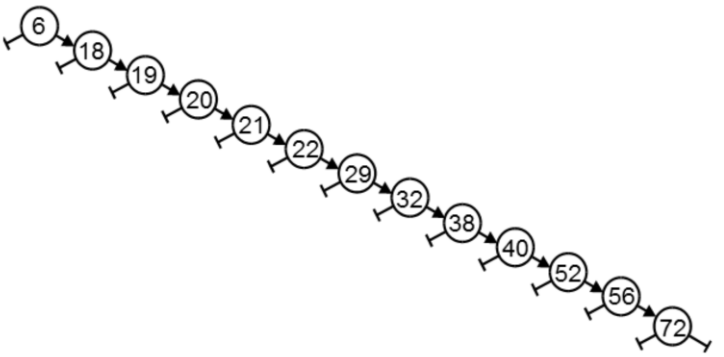
```
gnode * create_node(const char * name)
{
   gnode * p = (gnode *)calloc(1, sizeof(*p));

   if (p != NULL)
   {
       p->data = (char*)malloc(strlen(name) + 1);
       if (p->data)
          strcpy(p->data, name);
       else
          free(p), p = 0;
   }

   return p;
}
```

Using `calloc()` to allocate the memory for the node guarantees that the `child_count` data member is zero, and that the `children` pointer is `NULL`.

QACADV_v1.0

Unbalanced trees

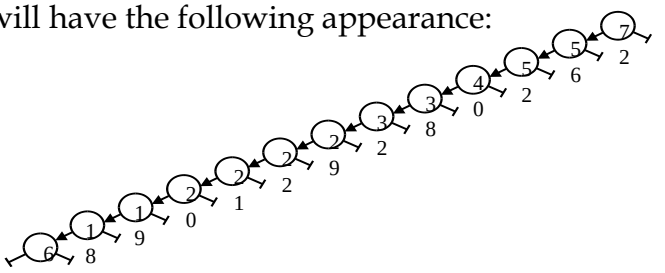
- The routines examined thus far can build "bad" (unbalanced) trees
- Consider creating a tree with the values 6, 18, 19, 20, 21, 22, 29, 32, 38, 40, 52, 56, 72



- Thirteen comparisons are now required to find that 98 is not in the tree

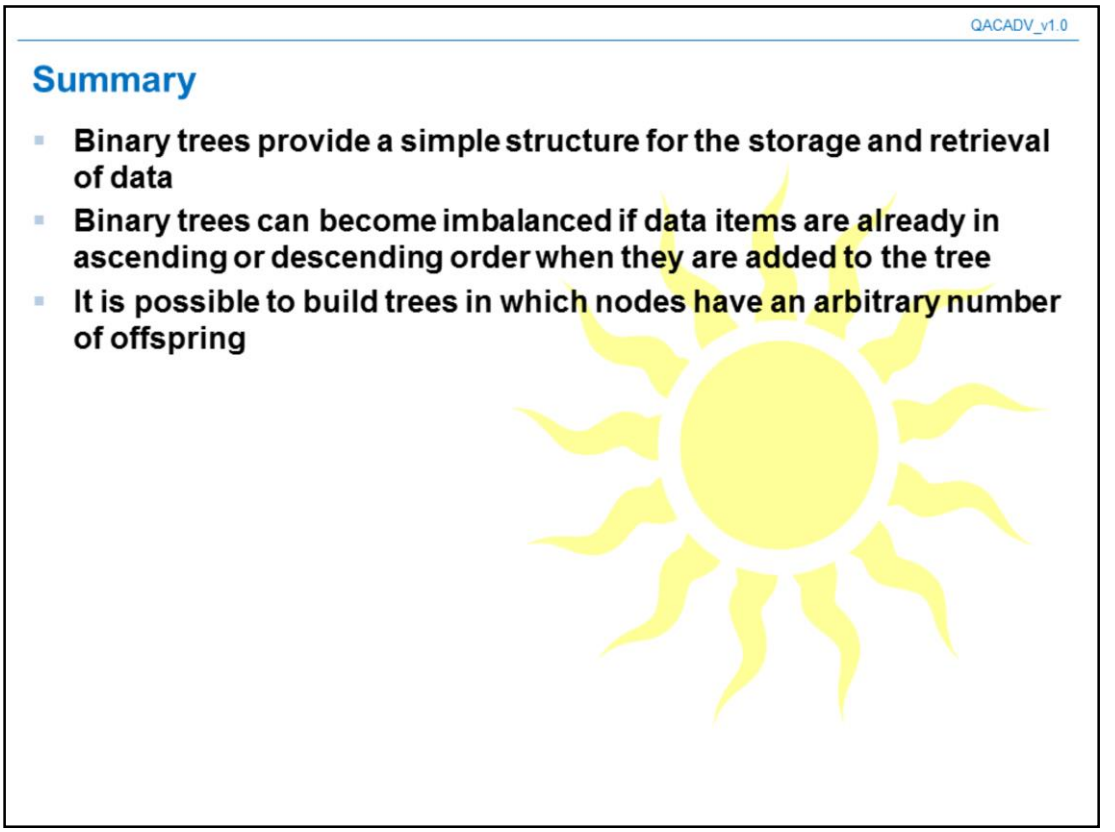
If we take the *same values as were used to build the tree in the previous example*, but insert the values in ascending order, the "tree" degrades into a simple linear linked-list.

Whereas the balanced tree required only four comparisons to discover that the value `98` was not present in the tree, thirteen comparisons are required in the "worst case" binary tree shown above. If the values are inserted in *descending* order, the tree will have the following appearance:



As before, a "worst case" tree is built. With randomly ordered data, adding n items to a tree results in a tree structure with a depth of $\log_2 n$. The *depth* of the tree is the maximum number of comparisons required when searching for a value. For example, inserting thirteen values in a tree should result in a tree with a depth of $\log_2 13 = 3.7$ (i.e. 4) levels. This value can also be found by simply finding the power of 2 which gives rive to a number greater than or equal to the number of levels, e.g. $2^4 = 16$.

If your calculator doesn't have a $\log_2$ button, use the log button (i.e. $\log_{10}$) and divide the answer by $\log_{10} 2$.



Binary trees are the most simple of all tree structures. A node may have zero, one or two *child* nodes. These child nodes are usually given the names `left` and `right`, implying an inherent order.

Adding data to a binary tree is straightforward. Many of the functions are implemented using recursion, reflecting the inherently recursive nature of a tree structure. Removing a node from a binary tree is more complicated because the tree has to be restructured when an item is removed. However, we have studied this issue in some detail during the chapter and the necessary code has been presented.

For a balanced tree, the number of comparisons required to find a particular data item will be *of the order of* log_2 of the total number of items in the tree. Thus, a tree containing a 100 items will require seven comparisons, whereas a tree containing 10000 items will require fourteen comparisons. *A tree containing 1000000 items will require only 20 comparisons!*

The worst case of a binary tree involves filling it with sorted data. This results in an imbalanced tree resembling a linked list, in which the number of comparisons becomes *equal* to the number of data items stored. In this case, a "tree" containing 1000000 items will require a number of comparisons *of the order 1000000!*

Some applications may require trees in which nodes can have an arbitrary number of child nodes. These may also be coded in C. The most efficient way to do this is to use dynamic memory allocation.