Chapter 2

Complex Data Structures

2.1 Introduction

One strength of the C programming language lies in the ability to abstract functionality away from the code that needs it. This is one fundamental advantage of using any of the higher level languages. . . in assembly, for instance, the nice `struct`ure code so frequently employed in C would be a mess of hard-coded offsets into memory that is difficult to keep track of by hand. The `struct` concept in C allows us to not only aggregate the simple data structures discussed in the last chapter, but to manage data which is much more complicated structurally. In this chapter, a few complex data structres will be explored.

2.2 N-ary Trees

An N-ary Tree is a very flexible data structure. N-ary nodes are one fundamental building block of a generalized graph strucure, but we will stick to the case of a directed acyclic graph (tree) for this discussion. An N-ary tree itself is a node with any number of children whose children are also N-ary trees. A tree can be N-ary and always have a constant number of children and still be N-ary, as long as the nodes **could** have any number of children. If instead it can only have 26 children, for example, it would be a 26-ary tree. It is also important to note that the number of children a node has does not depend on the number of children that its children have or that its parent has—they are separate and any such constraint would be dictated by the application, not the data structure.

Figure 2.1 shows the basic visual representation of an N-ary Tree. Notice that every node in the tree has three fundamental pieces of data:

- The **data** being stored in the node
- The **number of children** the node currently references
- The dynamic collection of **child references** themselves

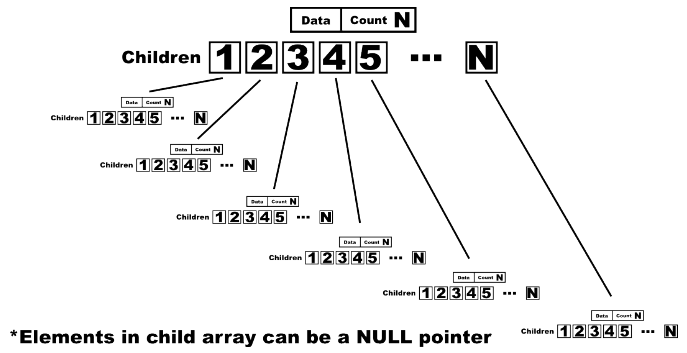

Figure 2.1: N-ary Tree Data Structure

```c
typedef struct sNaryNode {
             void   *data;   // Point to the node's data
     unsigned int    n;      // The number of children
  struct sNaryNode **child;  // The child list
} NaryNode;
typedef NaryNode NaryTree;
```

Code 2.1: N-ary Tree Structure Defintion in C

In C, there are many ways in which these can be represented. The data could be a statically defined data type in the structure. It could be a `typedef` that allows the programmer to specify the type before `#include`ing the library. In this document, the most generalized paradigm will be used: a void pointer (`void*`) will be used to point to any data in a generalized way. The `structure` definition for this is included in Code 2.1.

Semantically, a tree can be uniquely identified by its root. Unlike a linked list, whose head and tail can be tracked in a LinkedList structure to make linked list operations simpler, nearly all operations on a tree operate recursively, and as such it is useful to simply represent the tree with the root node. In this way, referencing a child node is identical to referring to that subtree. This useful property will be put to use soon.

Code 2.1 defines four things. First, it defines a structure to collapse all of the data that we need for a node (`data`, `n`, `child`) into a single memory map called `sNaryNode`, which is assigned to the type name `NaryNode`. This type name is also aliased to `NaryTree` for reasons which will be explained in the next section. Inside the memory structure of `struct sNaryNode`, the first field is `data`: this is a pointer to a memory location with an unspecified data type. As you will remember from Chapter 1, this pointer is simply a number that the computer understands to reference a specific place in its memory, and the void designation indicates that the type of data stored in that location will be specified by the programmer later when the data itself is actually desired. This gives us the flexibility to create a dynamic data structure which does not need

```c
NaryTree *createNode(int children, void *data)
{
  // Allocate space for a new NaryNode in memory
  NaryNode *node = (NaryNode*)calloc(1, sizeof(NaryNode));

  // Set the contents of the NaryNode appropriately
  node->data  = data;
  node->n     = children;
  node->child = (NaryNode**)calloc(children, sizeof(NaryNode*));

  // Return the node we initialized
  return node;
}

typedef void (*DataFreeFunc)(const void *);
void freeTree(NaryTree *tree, DataFreeFunc dFree)
{
  unsigned i;

  // Don't try this with a NULL pointer
  if (tree == NULL) return;

  // Free the children recursively
  for (i = 0; i < tree->n; ++i)
    freeTree(tree->child[i], dFree);

  // Free the child array
  free(tree->child);

  // Free the data if a function is provided
  if (dFree) dFree(tree->data);

  // And finally, free the structure
  free(tree);
}
```

Code 2.2: N-ary Tree Allocation and Freeing

to be reworked in order to use a new type of data. The structure contains a variable, `n`, that indicates how many children it is referencing. This is necessary for the tree to be truly N-ary, as it facilitates the requirement that each node can have any number of children.

2.2.1 Allocation and Freeing the Nary Nodes

When creating a dynamic data structure, the most fundamental operations which must be defined are its creation and removal. In C programming, this comes in the form of allocation and freeing of memory dynamically. Trees are defined recursively, so it is sufficient to create two functions to accomplish both of these tasks.

The code in Code 2.2 is fairly self-explanatory, but a few things bear mentioning. First, be sure to note the interchangeability of `NaryNode` and `NaryTree`.

```c
void *createIntData(int data)
{
  int *ptr = (int*)calloc(1, sizeof(int));
  *ptr = data;
  return ptr;
}

void *createDoubleData(double data)
{
  double *ptr = (double*)calloc(1, sizeof(double));
  *ptr = data;
  return ptr;
}

void *createStringData(const char *str)
{
  return strdup(str);
}

typedef struct {float x,y;} Point;
void *createPointData(float x, float y)
{
  Point *ptr = (Point*)calloc(1, sizeof(Point));
  ptr->x = x;
  ptr->y = y;
  return ptr;
}
```

Code 2.3: Example Data Allocation Functions

They will both be used in this book, depending on whichever data type makes more sense in context; this does not change the data in any way and is simply to make code easier to read. Second, notice that the data for the tree node must be passed into the `createNode` function as a `void*` already: this can get clunky, and may set off a red flag that this would be a good place to write a wrapper function for whatever datatype you are using, and you would be correct. A few examples of these are included in Code 2.3. Third, notice that the `freeTree` function does very little work for itself. It calls upon other instances of itself—the recursive nature of the tree data structure—to free themselves, then it simply frees the child array and calls the programmer-passed `dFree` function pointer to free the node's data if one has been provided before freeing the whole node. For most cases, simply passing `&free` will suffice for the `dFree` parameter, as this will simply free the data stored at `data`. All of the example data creation functions shown in Code 2.3 can be freed in this way. This will not be true in the rare instance where you only want part of the data freed or if the data points to a nested data structure whose children also need to be freed before the main structure itself to prevent memory loss.

2.2.2 Manipulating the Child Array

With the helper functions in Code 2.3, it is possible to do something as simple as the following:

```
NaryTree *tree = createNode(2, createIntData(10));
tree->child[0] = createNode(0, createIntData(5));
tree->child[1] = createNode(1, createIntData(15));
```

But that's not particularly fun or very N-ary. In order for it to be easy to manipulate the children, we need to be able to add and remove them at will. Dynamic memory allocation is our friend here, especially realloc. The child append function is simple enough, but the child insert function is a little more interesting. Make sure you understand it before you continue on, both how it works and how you might use it. The code for these functions is included in Code 2.4.

With these functions defined, it becomes possible to do far more with the N-ary tree code that makes sense on an intuitive level. For instance, to insert the points on a circle into an nary tree, and then generate inner points along the axes:

```
NaryTree *circle = createTree(0, NULL);
float rad; unsigned i, len;
for (rad = 0; rad < 2*M_PI; rad += M_PI/16)
  appendChild(root, createPointData(cos(rad),sin(rad)));
for (len = circle->n, i = 0; i < len; ++i)
  if (i % 8 == 0)
    appendChild(root, createPointData(
      ((Point*)root->child[i])->x / 2,
      ((Point*)root->child[i])->y / 2
    ));
```

The advantage in this particular situation is that the coder doesn't need to precalculate how many points are necessary, and the code can adjust automatically should the number of necessary points change. This sort of paradigm also works doubly well when the user does not know how many data points will be necessary. For instance, when writing a generalized string processing algorithm, setting a fixed upper bound for number of child nodes would limit the flexibility of the algorithm. Using an N-ary node removes this limitation, as well as minimizing the extraneous memory being used.

```
int appendChild(NaryNode *root, void *data)
{
  // Increment the degree of the node
  root->n++;

  // Reallocate the child array (n children in the child array)
  root->child = (NaryNode**)realloc(root->child,
      (root->n)*sizeof(NaryNode*));

  // Add the newNode into the child array and increment degree
  root->child[root->n - 1] = createNode(0, data);

  // Return the index of the child we just inserted
  return root->n - 1;
}

int insertChild(NaryNode *root, int idx, void *data)
{
  unsigned i;

  // First, we make space
  root->n++;
  root->child = (NaryNode**)realloc(root->child,
      (root->n)*sizeof(NaryNode*));

  // Then we rotate everything back by one and insert data
  for (i = root->n-1; i > idx; --i)
    root->child[i] = root->child[i-1];
  root->child[i] = createNode(0, data);

  // Return the index of the child we just inserted
  return i;
}

void deleteChild(NaryTree *root, int idx, DataFreeFunc dFree)
{
  unsigned i;

  // Delete the child
  freeTree(root->child[idx], dFree);

  // Remove the defunct child
  for (i = idx; i < root->n - 1; ++i)
    root->child[i] = root->child[i - 1];

  // And finally, reallocate
  root->n--;
  root->child = (NaryNode**)realloc(root->child,
      (root->n)*sizeof(NaryNode*));
}
```

Code 2.4: Child Insertion and Deletion