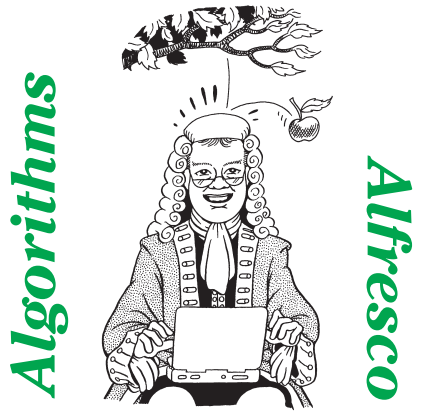
Dear Sir Or Madam

Julian responds to readers' emails



by Julian Bucknall

Writing a monthly column seems a simple job, but for me it's a schizophrenic occupation. Sometimes the text and code flow like magic: it's like driving a Volvo 1800S in the back lanes of the Cotswolds. At other times you have the handbrake on and black smoke billowing out of the back. In the former case everything seems heaven-sent: the description and implementation come together beautifully and I get a terse 'Reads well' from Our Esteemed Editor. In the latter, it's hard work, the deadline looms, Knuth is obtuse, the code won't work, my wife Donna gives up and puts my dinner in the oven. It's with these articles I get the dreaded readers' emails: 'X won't compile with $R+', 'You haven't explained Y very well and I've no idea how to use it', etc.

Writing can be fairly glamorous. People come up to me at the Borland Conference and want to shake my hand and say that they enjoy my articles; I get email saying that I've saved their reputation with their boss because they could use the code in *Algorithms Alfresco*. This latter kind of contact with my readers is extremely gratifying, because in the final reckoning the main reason I write these articles is to help other developers get their jobs done.

No matter which type of email I get, I always respond (sometimes later rather than sooner), but I get the feeling that I'm not helping enough. By replying to a single person that algorithm X is best used in situation Y, by engaging in an email correspondence about the tricky aspects of data structure Z with someone, I'm helping this one developer, but the other readers don't get to participate in the conversation.

This article, then, is a précis of several readers' emails, together with my comments and enhanced code. The disk this month contains several bug-fixed and enhanced *Algorithms Alfresco* units, so be prepared to update your development folders. If you have written to me in the past after a particular article, you may recognize your question and my answer: if so, thank you for emailing me. This way I get a feeling for how well I'm doing and how good the articles are for your development needs.

Issue 57 (May 2000)

Hans: I guess I won't be the only complaining one but in listings 3 & 4 of your article you've found quite a nice way to get round the millennium problem. I guess if you'd used `MOD` *instead of* `DIV` *the code might even produce the results you were looking for...*

Ouch. *Mea culpa*. Hans was completely correct. The column concerned was on performance. I'd rushed off a couple of routines to calculate the number of days in a month, the first being pure code with lots of `if` statements and the second making use of an internal table: the intent being to show that the second was faster. For the month of February the routines had to calculate whether the year concerned was leap or not. I used `DIV` in the calculation instead of `MOD`, thereby producing a nonsensical result. The test for a leap year should be:

```
Leap := ((Year mod 4) = 0) and
  (((Year mod 100) <> 0) or
  ((Year mod 400) = 0));
```

Issue 44 (April 1999)

Lasse: I thought I'd just let you know of a better way to store the Huffman tree in the compressed data stream, at least better than the two methods you discussed in the article. Instead of storing the character counts you should store the actual tree.

The column Lasse was talking about discussed Huffman data compression. The problem, if you recall, is that you need to transmit the Huffman tree with the encoded data, since you can only decode the data using the tree, and the tree is unique to that particular data distribution. I'd proposed two possible ways of tacking on the tree onto the compressed data. The first (algorithm I) was to add a 256 `longint` table of character counts. Using this information it is possible for the decompressor to build the tree in exactly the same way that the compressor did. However, 256 `longints` is 1,024 bytes, which seems excessive to add to a compressed data file. My alternative plan (the one I used in the code on the disk) was to output the character counts 'DFM-style' (algorithm II). That is, if the count were 255 or less, output a single byte value to signify 'the next value you read is a single byte', and then output the count as a byte. If the count were between 256 and 65,535 inclusive, output a single byte value to read 'the next value you read will be a word (two bytes)' and then output the count as a word. If the count were any greater than that, output another byte value, and then the count value as a `longint`.

Lasse pointed out a much better scheme, one that I hadn't seen before. It is succinct, very compressed and very clever. The only reason I output the character counts was so that the decompressor could rebuild the Huffman tree. After that, there was no need for the character counts. Lasse argued that the compressor should output the *tree* itself.

Recall that in a Huffman tree, the nodes divide themselves into two types: internal nodes, which

always have two child nodes (be they internal or external or both) and which are empty (that is, they contain no data), or external nodes that hold a single character (and, of course, by definition these have no children). So the plan is to traverse the tree, visiting the nodes one by one; we output a single bit to say whether a node is internal or not and, if external, the character in that node. The routine is recursive, but, since the tree is bounded (there are at most 256 external nodes and hence 255 internal nodes, and the maximum path from root to leaf is 256 links), we shan't be in any danger of blowing the program stack.

Suppose we have a routine called `WriteNode` and we pass a node to it as a parameter (we start off the process by passing the root node, of course). If the node passed in is external, write a set bit to the bitstream and then write the character in that node as 8 bits. If the node passed in is internal, write a clear bit to the bitstream. Next, if the node is internal, make a recursive call to `WriteNode` passing the left child, and then a recursive call to `WriteNode` again, this time passing the right child. And that's all there is to it.

Of course, writing a tree is all very fine and dandy, providing that it is possible to read it again. We write another recursive routine, this time called `ReadNode`. This routine reads a bit from the bitstream. If the bit is set, we need to create an external node, and then read the character for that node from the bitstream (8 bits). If the bit is clear, it's an internal node. We recursively call `ReadNode` for the left child (we assume `ReadNode` returns the node it had to create, so that we can make the link), and then for the right child.

The algorithm for encoding the tree encodes *only* the tree: a character that does not appear in the uncompressed document will not be in the tree and therefore will not be encoded by the write routine. Hence we can calculate exactly the number of bits required to store the tree. If the number of different characters in the original document were n (with a maximum of 256, of course), then the number of external nodes is also n, occupying $9n$ bits, and the number of internal nodes is n-1, occupying n-1 bits. The total number of bits needed to store the tree is therefore $10n$-1. For the worse case scenario, n is 256 and therefore the tree would take 2,559 bits in the compressed stream. 2,559 bits is 319 bytes with 7 bits left over. Compare that with either 1,024 bytes in my algorithm I, or a minimum of 512 bytes (every character count is less than 255) to a maximum of 1,280 bytes (every count is greater than 65,535) with algorithm II. A pretty good improvement.

Listing 1 shows the `WriteNode` and `ReadNode` routines for the Huffman compression. The original code used a preset array of nodes for the Huffman tree, and the individual nodes are represented not by pointers but by indexes into this array. This month's disk has the complete Huffman implementation with this improvement.

A further question might be asked: can this algorithm be used to store *any* binary tree? Well, with some modifications, yes. In an off-the-shelf binary tree (rather than the custom, very specialized, Huffman tree), a node can have no children, a left child only, a right child only, or both children, and a node can contain its own data in some form or other (a string, an object, a record). This means that we have 8 possible types of node: a leaf with no data, a leaf with some data, an internal node with a left child and no data, an internal node with a left child and data, and so on. These possibilities can easily be represented by a single byte as an enumeration.

The plan is the same as before. To store the entire tree we write a recursive `WriteNode` routine, passing to it a node and the stream to which the tree is to be written. We start the whole process off by calling `WriteNode` with the root of the tree. The `WriteNode` routine works out the type of node passed to it (has it any data? which children are present?) and writes out the correct enumeration value to the stream. If it has any, it then writes out its data to the stream. Then, if it has a left child, it recursively calls `WriteNode` with this child. On return from this call, if it has a right child it recursively calls `WriteNode` with this child.

With a general binary tree we'd have to consider runaway recursion, in other words that the tree is so deep that we'd blow the program stack. If we did have the possibility of running across such a tree, we would have to rewrite the `WriteNode` routine to remove recursion. However, this possibility is fairly remote, especially for a tree that we're attempting to store in a stream, so leaving `WriteNode` as a recursive routine is acceptable.

`ReadNode`, of course, is a recursive function as before. On

➤ *Listing 1: WriteNode and ReadNode for a Huffman tree.*

```
procedure WriteNode(aStream : TOutputBitStream; aHTree :
  PHuffmanTree; aNodeInx : integer);
begin
  {for a leaf, write a 1 bit, followed by the character}
  if (aNodeInx < 256) then begin
    aStream.WriteBit(true);
    aStream.WriteByte(aNodeInx);
  end else begin
    {for an internal node, write a 0 bit, then the left
     subtree, then the right subtree}
    aStream.WriteBit(false);
    WriteNode(aStream,aHTree,aHTree^[aNodeInx].hnLeftInx);
    WriteNode(aStream,aHTree,aHTree^[aNodeInx].hnRightInx);
  end;
end;
function ReadNode(aStream : TInputBitStream; aHTree :
  PHuffmanTree; var aMaxInx : integer) : integer;
var IsLeaf : boolean;
begin
  {read next bit to determine which node we have to create}
  IsLeaf := aStream.ReadBit;
  {if is a leaf return its node index (ie the character)}
  if IsLeaf then
    Result := aStream.ReadByte
  {if it's internal node, get the left and right subtrees}
  else begin
    inc(aMaxInx);
    Result := aMaxInx;
    aHTree^[Result].hnLeftInx :=
      ReadNode(aStream, aHTree, aMaxInx);
    aHTree^[Result].hnRightInx :=
      ReadNode(aStream, aHTree, aMaxInx);
  end;
end;
```

```
procedure TaaExpressionParser.epFormRPNSubExpr(aOp : char;
  aCharPos : PChar);
var
  PrecOp   : integer;
  PrecTop  : integer;
  TempOp   : char;
  Operand1 : string[255];
  Operand2 : string[255];
begin
  {this routine is called when the operator about to be
    pushed, aOp, has a precedence lower than or equal to the
    operator on top of the operator stack. We need to pop
    off some operators and operands and form some RPN
    expressions to push onto the operand stack, until the
    operator stack is exhausted or the top operator has a
    precedence value less than the given operator's
    precedence value.}
  PrecOp := epGetPrecedence(aOp);
  PrecTop := epGetPrecedence(FOpStack.Examine);
  while (PrecOp <= PrecTop) and (PrecTop > 1) do begin
    TempOp := FOpStack.Pop;
    if (TempOp = UnaryMinus) then begin
      if (FStStack.Count = 0) then
        epRaiseBadExpressionError(aCharPos);
      Operand1 := FStStack.Pop + UnaryMinus;
      FStStack.Push(Operand1);
    end else begin
      if (FStStack.Count < 2) then
        epRaiseBadExpressionError(aCharPos);
      Operand2 := FStStack.Pop;
      Operand1 := FStStack.Pop + Operand2 + TempOp;
      FStStack.Push(Operand1);
    end;
    if FOpStack.IsEmpty then
      PrecOp := 0
    else
      PrecTop := epGetPrecedence(FOpStack.Examine);
  end;
  {if the given operator was a right parenthesis the top of
    the operator stack *must* be a left parenthesis and we
    should remove it}
  if (aOp = ')') then begin
    if FOpStack.IsEmpty or (FOpStack.Examine <> '(') then
      epRaiseBadExpressionError(aCharPos);
    FOpStack.Pop;
  end;
end;
```

➤ *Listing 2: Expression-to-RPN parser.*

entering the routine we create a node. We read the enumeration value from the stream and this will tell us how many children we have of what kind, and whether we have any data. If there is data to be read, we read it. If we're supposed to have a left child, recursively call the `ReadNode` routine; similarly for the right child. We finally return the node we created.

Knowing the readers I have, I'm sure you could extend this to work with a red-black tree. Using such a self-balancing binary search tree, it would not be strictly necessary to store the tree as is: we just need to store the data. On reading the stream again, we could merely insert each data item into a brand new tree. We wouldn't get exactly the same tree, of course, but the important ordering property would still be satisfied. However, this algorithm is an O($n$log(n)) operation, whereas, at the expense of a single byte per node we could turn it into an O(n) operation.

Issue 46 (June 1999)

John: I have a problem with your RPN expression parsing routines, relating to operators with the same order of precedence. For example, the expression '1-2+3' gets parsed to '123+-' which evaluates to -4. As '+' and '-' have the same precedence, they should be evaluated in left to right order, giving '12-3+' instead, evaluating to +2.

Oh boy, John really nailed me on this one, good and proper. He's right, of course, and it's my fault for not testing enough. The article he refers to is the one where I showed how to parse an arithmetic expression into its reverse Polish notation form, which is extremely easy to evaluate. Unfortunately, I didn't consider all the ramifications of the parsing technique.

Let's recap the procedure as I originally outlined it, using John's example. Suppose we have two stacks, one for numbers and one for operators. We read the expression '1-2+3' from left to right. Get the '1', push it on the number stack. Get the '-'. We don't know what to do with this yet since we don't have the other operand, so push it on the operator stack. Get the '2'. We *could* now evaluate the '1-2' but we're not sure at this moment whether the next operand is of higher precedence or not (for example, in the expression '1-2*3'), in which case the operand belongs to it rather than the '-', so we push it on the number stack to get it out of the way. Get the '+'. The rule I stated is: if you've got an operator token, peek at the top of the operator stack. If it is of higher precedence then the one we've just read, you immediately form an RPN expression with the top operator and push that onto the number stack. You continue with this process until the top of the operator stack is of lesser or equal precedence, in which case you push the operator just read. So, we peek at the top item on the operator stack to see if it has a higher precedence than '+'. It doesn't so we push it. Get the '3', and push it on the number stack. At this point, we've run out of expression and so we start popping stuff off the stacks. Pop the '+'. This requires two operands, so pop them off ('3' then '2') and form an RPN expression, '23+' and push it onto the number stack. Pop off the next operator, '-', which requires two operands ('23+' and '1') and form an RPN expression, '123+-'. Bzzzt! Wrong, and thank you for playing.

The obvious thing is that the rule I formulated was wrong. What it *should* have said is this. Get the next token. If it is an operator, peek at the top of the operator stack (if there is anything on the stack, of course!). If the operator there is of greater or equal precedence, then form an RPN expression with it and as many operands as it needs and push the result on the number stack. Peek at the top of the operator stack again, and repeat the same process if the operator has equal or higher precedence. Continue like this until the stack either empties, or the operator is of lesser precedence. Using this new rule, we indeed get the correct answer of '12-3+'. Listing 2 shows the revised expression-to-RPN parser.

Issue 46 (June 1999)

John: Not so fast! Listing 4 in the same article has a more serious memory overread problem! You are trying to read a complete number or identifier and possibly managing to read beyond the end of the expression string.

[Sound effect: Head hitting desk repeatedly] Er, yes, John, you are

quite right again. The original code had this as it tried to read a complete number:

```
while Expr[i+1] in NumberSet
  do begin
  OperandSt :=
   OperandSt + Expr[i+1];
  inc(i);
end;
```

There should be a check in there to make sure we don't go off beyond the end of the `Expr` string. If the memory at the end of the string happened to have ASCII digits in it, the while loop could go on for quite a while. Listing 3 has the debugged RPN expression evaluator. (Please let me extend many thanks to John Leavey for bearing with me on this quite atrocious bit of design and coding.)

Issue 54 (February 2000)

Paulo: I read with interest your article on encryption, but I found one historical inaccuracy: The ENIGMA machine was actually broken by a group of Polish mathematicians in the 1930s as described in The Code Book by Simon Singh. It was only because Rejewski showed that it was possible to do it that the English took on the task after that...

Paulo is absolutely correct. I completely minimized the contribution made by the Poles to the cracking of the ENIGMA machine. They cracked a simpler ENIGMA machine in the 30s, leading the British to assume that a more complex ENIGMA was also crackable and thereby going ahead and trying to do it. The British mathematicians also had some good luck in that the German operators were not loath to cut corners in setting up the machine, leading to a point of attack.

I haven't read Singh's cryptography book that Paolo mentions, but I did enjoy his earlier one on the proving of Fermat's Last Theorem and if *The Code Book* is as good as its predecessor, it must be a good read.

Issue 42 (February 1999)

Martin: When working with linked lists, for example, you define records with pointers to them and then allocate memory when you need a new node. Why do you do that, rather than define an object class? With an object, the constituent properties are already dereferenced, thus avoiding the '^' that my managers (here) hate to see. In my ignorance, the use of object classes seems to fit in more with the Delphi OO model too. Am I missing something?

It's all a matter of preference, I suppose. Let's take the double linked list as an example. We have to have nodes with both a forward link and a backward link to other nodes, and we must have data with each node. As I see it, we have two choices of how to implement this linked list, as shown in the code fragment below:

```
PNode = ^Tnode;
TNode = record
  Next  : PNode;
  Prior : PNode;
  Data  : ..anything..
end;
TNode = class
  Next  : TNode;
  Prior : TNode;
  Data  : ..anything..
end;
```

Now, if we were writing a linked list from scratch just for the data we wanted to put in it, there's no real difference between the two. The class definition fits better into the Delphi model if you like, that I won't argue.

Now look at it from a different viewpoint: we want to write a reusable linked list class. A class we can use all over the place for storing wildly different types of data. Sometimes they'll be objects, yes, other times, they'll be strings, yet other times you'll have some record structure. In this case I would argue that having the `Next` and `Prior` references visible is detrimental to our use of the linked list ('Hey, I want to have a linked list of `TButtons`. Are you saying I must create a `TButton` descendant that has `Next` and `Prior` references, then add this descendant to the component palette, and always remember to use it? No way, José').

So, we agree that in this 'reuse' case we need to have the node information separate from the data information. The node will

➤ *Listing 3: Evaluating an RPN expression.*

```
function TaaExpressionParser.epGetValue : double;
var
  DblStack : TaaFloatStack;
  i        : integer;
  Operand1, Operand2 : double;
  Expr     : string[255];
  OperandSt: string[255];
begin
  if not FParsed then
    epParseToRPN;
  DblStack := TaaFloatStack.Create;
  try
    {read through the RPN expression and evaluate it}
    Expr := FStStack.Examine;
    i := 0;
    while (i < length(Expr)) do begin
      inc(i);
      if (Expr[i] = ' ') then begin
        if (Expr[i+1] in NumberSet) then begin
          OperandSt := '';
          while (i < length(Expr)) and
            (Expr[i+1] in NumberSet) do begin
            OperandSt := OperandSt + Expr[i+1];
            inc(i);
          end;
          DblStack.Push(StrToFloat(OperandSt));
        end else begin
          OperandSt := '';
          while (i < length(Expr)) and
            (Expr[i+1] in IdentifierSet) do begin
            OperandSt := OperandSt + Expr[i+1];
            inc(i);
          end;
          DblStack.Push(FVarList.Value[OperandSt]);
        end
      end else begin
        if Expr[i] = UnaryMinus then
          DblStack.Push(-DblStack.Pop)
        else begin
          Operand2 := DblStack.Pop;
          Operand1 := DblStack.Pop;
          case Expr[i] of
            '+' : DblStack.Push(Operand1 + Operand2);
            '-' : DblStack.Push(Operand1 - Operand2);
            '*' : DblStack.Push(Operand1 * Operand2);
            '/' : DblStack.Push(Operand1 / Operand2);
            '^' : DblStack.Push(Power(Operand1, Operand2));
          end;{case}
        end;
      end;
    end;
    Result := DblStack.Pop;
  finally
    DblStack.Free;
  end;
end;
```

have an opaque pointer to the data (it could be a pointer to a record structure, a string, or an object, we don't care). The node would have to have the following information:

```
TNode = record or class
  Next  : PNode or TNode;
  Prior : PNode or TNode;
  Data  : pointer;
end;
```

And then we could build up a reusable linked list class quite easily without the user having to worry about maintaining the `Next` and `Prior` pointers, etc. So should it be a class or a record, this node? It doesn't really matter at a conceptual level, but it *does* matter at an efficiency level. If the node is a record, we can allocate them in blocks of a hundred or more and use a node manager to dole them out when needed. Objects will have to be allocated individually (there's funky stuff going on during the constructor). My tests have shown that using a node manager is about 4 times faster than single node allocations from the heap. Ergo, I use records.

Issue 51 (November 1999)

Alan: I enjoyed your article on floating points and the alignment and misalignment stuff. I thought you might be interested in the following quotation from Borland's Danny Thorpe: 'Delphi 5's alignment padding now supports 8 byte alignment as well. Data types 8 bytes or larger will be aligned to 8 byte boundaries.'

As soon as I received Alan's email, I checked it out for myself. It's true, Delphi 5 does align double variables correctly, and as a result applications that make heavy use of floating point operations are 6% to 10% faster. Hence, in Delphi 5, you don't have to worry too much about misaligned double variables since the compiler will take care of most of it for you. This even extends to local double variables: Delphi 5's compiler guarantees that they will be 8-byte aligned for maximum efficiency. The `Extended` type does still suffer by comparison, of course: being 10 bytes long just isn't efficient.

Issue 50 (October 1999)

Dave: We want to use the red-black tree, but we need to insert many, many nodes as efficiently as possible. It is also very likely that the raw data we insert will have duplicates. If the data has already been inserted into the tree, we need to retrieve the data from the tree to update it, if not then we need to merely insert the data. Reading your article and the code, it seems that you intended us to do the following:

```
find the item
if found then
  update the data
else
  insert the item
```

The problem that I see is that the insert operation performs a find operation as well, in order to work out where to put the item. So, because we have many duplicates, we are essentially reduced to performing a find operation twice for each insert. For several hundreds of thousands of nodes, this duplicated find time is likely to add up.

Dave's reasoning is spot on the money and was well investigated. `Insert` *has* to do a find operation in order to work out where to put the new item. The find will fail, of course, but it will fail exactly at the point where the item should be inserted, so we can't do without it. Of course, I was being very 'pure' in my original design: a `Find` does a search, whereas an `Insert` adds the item to the tree. However, Dave's requirement is very common: try and insert an item, but if it already exists, return the original. (I had essentially the same message from someone else who was using EZDSL's hash table.)

So we need to enhance the original binary search tree and the red-black tree to perform this `InsertOrGet` algorithm. To avoid duplicating code I had to separate out the balancing algorithm from the red-black tree insertion algorithm since it would now be called from two different places. The other changes are not that great either. Listing 4 shows the new `InsertOrGet` method for both the binary search tree and the red-black tree. It neatly implements Dave's requirement.

On Compiler Options

Man In Street: When I compile your code with such-and-such a compiler option it doesn't compile or it raises an exception when run.

This kind of question comes up not only with my *Algorithms Alfresco* columns but also every now and then with TurboPower's libraries. The favorite compiler options raised are `{$R+}` (range checking) and `{$Q+}` (arithmetic overflow checking), although I once had to write a reasoned reply to a customer that TurboPower was not going to make sure that its code would compile with `{$X-}`.

First things first, I recognize that these people have a point. I'm not going to dismiss them out of hand, but I am going to try and explain my reasoning for using the compiler options I do. I'm going to refer to Delphi 5 exclusively, by the way.

Speaking personally, I divide up the compiler options into two camps: The Must-Not-Be-Changed options, and the User-Can-Change-If-Required options.

The first collection of options is, in my view, cast in stone. It would be perverse indeed to change any of them. I always use short-cut Booleans `{$B+}` and extended syntax `{$X+}` (could *you* live without proper `PChars`?), always disable var string checking `{$V-}`, always raise exceptions for I/O errors `{$I+}`. I used to use the `$A` alignment compiler option, but these days I prefer the control of using packed records and organizing the alignment myself. Huge strings `{$H+}` are *de rigueur* these days. Assignable typed constants `{$J+}` just shows my ancestry: I'm sure I can code to avoid them, but I'm stuck in my ways. I dislike the typed @ operator option so I always leave it as `{$T-}`.

The rest, I suppose you could say, are up for grabs. Open string parameters `{$P+}` are pretty useless these days with long strings. Optimization `{$O+}`, stack frame `{$W+}`, stack checking `{$S+}`, debug info `{$D+}` and local symbols `{$L+}` are basically debug versus release

```
function TaaBinarySearchTree.bstInsertPrim(aItem : pointer;
  var aExists : boolean; var aUseLeft : boolean) :
  PaaBTNode;
begin
  {first, attempt to find the item; if found, return it}
  if bstFindItem(aItem, Result, aUseLeft) then
    aExists := true
  else begin
    {otherwise, this returns a node, so insert there}
    aExists := false;
    Result := FBinTree.InsertAt(Result, aUseLeft, aItem);
    inc(FCount);
  end;
end;
procedure TaaBinarySearchTree.Insert(aItem : pointer);
var
  UseLeft, WasFound : boolean;
begin
  bstInsertPrim(aItem, WasFound, UseLeft);
  if WasFound then
    raise Exception.Create('TaaBinarySearchTree.Insert: '+
      'duplicate keys not allowed');
end;
function TaaBinarySearchTree.InsertOrGet(aItem : pointer;
  var aCurItem : pointer) : boolean;
var
  UseLeft, WasFound : boolean;
  Node     : PaaBTNode;
begin
  Node := bstInsertPrim(aItem, WasFound, UseLeft);
  Result := not WasFound;
  aCurItem := Node^.btData;
end;
procedure TaaRedBlackTree.Insert(aItem : pointer);
var
  Node     : PaaBTNode;
  WasFound, UseLeft  : boolean;
begin
  {insert the new item, get back the node that was inserted
    and its relationship to its parent}
  Node := bstInsertPrim(aItem, WasFound, UseLeft);
  if WasFound then
    raise Exception.Create('TaaRedBlackTree.Insert: '+
      'duplicate keys not allowed');
  {balance the tree}
  rbtBalanceAfterInsert(Node);
end;
function TaaRedBlackTree.InsertOrGet(aItem : pointer;
  var aCurItem : pointer) : boolean;
var
  Node     : PaaBTNode;
  WasFound, UseLeft  : boolean;
begin
  {insert the new item, get back the node that was inserted
    and its relationship to its parent}
  Node := bstInsertPrim(aItem, WasFound, UseLeft);
  aCurItem := Node^.btData;
  Result := not WasFound;
  {balance the tree, if inserted}
  if Result then
    rbtBalanceAfterInsert(Node);
end;
```

➤ *Listing 4: Insert or Get with a binary search tree.*

code options: they're only there to help debugging, and once the code is fully tested they can be toggled. And that leaves us with range checking and arithmetic overflow checking.

Range checking has admirable reasons for existing: catch the off-by-one errors, check for an array index being out of bounds, and so on. Great if you always declare your arrays to be exactly the right size. However, I find myself using arrays that vary in size: I declare a pointer-to-an-array type and the array type itself is deliberately made as large as possible. Not because I'm ever going to declare a variable of that type, of course, but so that I can dynamically size a pointer to that type. If I want 100 elements, I `GetMem` 100 elements. Range checking won't help me with this type of array. One of the most useful types in Delphi's VCL is the `PByteArray`. Cast any pointer to a `PByteArray` and suddenly you can read the memory the pointer points to, byte by byte. Range checking wouldn't work with this. Now, I agree that range checking does have its place, don't get me wrong, but it is not the universal panacea many people believe it to be. As a matter of course I have it on all the time during development, but I have not had a range check error in months, if not years. Memory overwrites and overreads by the dozen in that time of course, but range checking didn't catch them.

Arithmetic overflow checking? I hate it, mainly because it's not implemented correctly. If I'm manipulating bits in a byte, not doing arithmetic at all, I get arithmetic overflow errors. If I'm manipulating bits in a longint in the same manner, I don't get overflow errors at all. Go figure. (I've reported this one as a bug: it's inconsistent and I wasn't doing arithmetic, I was bit-twiddling. Compression and encryption can be very hard to do without bit-twiddling. But I digress.) But, I'm trying to keep it on to see what errors it catches. Apart from my bit-twiddling in compression, zip so far.

Issue 53 (January 2000)

So who won?

Yes, indeed! Who? I've finally got round to checking the entries for my simulated annealing competition. I had five entries in all, of which one, unfortunately, had to be disqualified for using another algorithm entirely.

The entrants were: Lars Hjelmi (who managed to get me a solution before I even received my copy of that issue of *The Delphi Magazine*!), Franz-Leo Chomse (close on his heels, and who said that he'd managed to code it so quickly because he was waiting for a new version of TurboPower's FlashFiler), Sergey Kostinsky (who produced a non-annealing version that was very fast), Gert Kello (who gave me a well done GUI application with several options to try out), and Gavin Clements (who gave me one program and then, a couple of weeks later, a faster one).

Franz-Leo approached the problem as a knapsack. His approach, basically, was to find every single word in every possible combination in the letter matrix, and then to stuff the knapsack. He maximized the count of the number of words. Unfortunately, that meant that a word could be duplicated and indeed the final solution (16 words) had four separate TOs and LAZYs. Rereading my instructions, I didn't expressly discount this possibility.

Onto Lars. He approached it the same way (precalculating all the possible word paths) and maximizing the number of words in a knapsack. The app allowed you to control the number of iterations and whether to accept duplicate words. 15 words in a fast time.

Sergey's app, although discounted because he didn't use simulated annealing, was very fast. Unfortunately, the best answer it came up with only had six words.

Gert's solution was a nicely written GUI app, with lots of options to try out (including an `About` box!).

Also a great class hierarchy, and well designed. Another maximize the wordcount solution. And it was very good at solving too: it produced 15 words at the drop of a hat, a slightly different set to Lars though.

Talking of nice GUI apps, Gavin's solution was a well-worked example. However, he decided to search for any word in the WORD.LST file that I supplied with the December 1999 issue, rather than the list of words I supplied with the *Simulated Annealing* article (it provided the best display of the answer, though). However, when I ran it with the correct word list, it didn't find a solution at all, which was very bizarre.

To be honest, all of these solutions were excellent. I was very impressed with their design, code and approach (if any were permanent US residents, I'd offer them a job at TurboPower). Deciding the winner was very difficult, but I said that I'd award the prize to the fastest, and Lars' solution edged ahead of Gert's, but only just. Well done all entrants, and especially Lars. His word list:

```
"XYZZY" =  E6 E7 F7 E8 F8
"TO"    =  L3 K4
"LAZY"  =  D6 B7 C7 A8
"MICE"  =  K1 K2 L2 K3
"ALL"   =  B4 A5 B5
"SONG"  =  J3 I4 H5 G6
"TWO"   =  G4 H4 G5
"DOG"   =  E1 D2 D3
"GOOD"  =  I5 H6 I6 H7
"FIVE"  =  H3 I3 J4 K5
"THE"   =  K6 J7 L7
"BEST"  =  K7 J8 K8 L8
"THREE" =  A1 B2 A3 B3 C3
"MEN"   =  H2 I2 J2
"LOVER" =  F1 G1 H1 F2 G2
```

Summary

I hope you've enjoyed reading these questions and answers, and that you're left with some snippet that helps you in your work. I shall be doing this again in the future, providing that, of course, you send me thought-provoking emails or I introduce some egregious bug. Next month, a new occasional feature that I think you'll like: *Applied Algorithms Alfresco*. I get some emails that say something like this: 'I want to implement this particular feature in my application, but I haven't a clue how to go about it, or, despite reading every article you've written, which data structure or algorithm to use.' Some of these answers would fit amongst others in an article like the one you are reading now, some of them would require an article of their very own. So, next month, the first applied algorithm article. See you then!

Julian Bucknall can be reached at julianb@turbopower.com. The code that accompanies this article is freeware and can be used as-is in your own applications.