Under Construction: Website Indexing, Part 1

by Bob Swart

This month, we continue our Delphi internet solutions coverage with the first of a two part series on the implementation of a mini website search engine, starting with a website page scanner, index generator and (single) keyword finder, called IndexBob.

If you've ever visited Alta Vista or Yahoo you already know what a search engine is: a place where you can specify keywords in an editbox (sometimes using `AND`, `OR`, `NOT` and `NEAR`) and a `Search` button that starts a process of looking in a big database for you, coming back with a list of webpages (URLs) that contain these keywords. Usually this takes some seconds and results in a large number of hits, which is why some search engines also have information to help people specify the most effective way to construct search queries.

Another phenomenon is a dedicated (local) website search engine that you can only use to search in the sub-webpages of one website. Microsoft has something like this on their website, where you can search for keywords on all or part of their website, and so have many other organisations (like yours when you've finished reading this article). This kind of mini search engine offers welcome support for visitors to a website and should be placed on the main page (so visitors will see it right away).

In this two-part article we'll build our own local website search engine.

Website Search Engine

Let's first try to image what a website search engine would look like. Basically, there are two approaches: dynamic and static. The dynamic approach involves active searching (parsing) of some or all the pages in the website, looking for a set of keywords, at the time the user requests the search. This takes time, but at least you know you're searching the real pages. The static approach involves generating an index of all the keywords found in the webpages. This means we need a website indexing program to build the index. Searching for keywords then consists of looking them up in the index and returning the webpages associated with these keywords. This is way faster than the dynamic approach, but we must take care to keep the index up to date. For that reason, this latter approach is best suited to relatively quite websites, or those where the webmaster is the only one to make updates, after which s/he can run the index program again to re-index the entire website. For an open company intranet, for example, where every employee can update pages, the static solution would likely be unsuitable.

For my own website (which is at www.drbob42.com if it's not already on your favorites list!) and many others the static approach is ideal, as long as the webmaster remembers to re-index the site after each update! So, I've decided to implement the static approach in this article.

Website Scanning

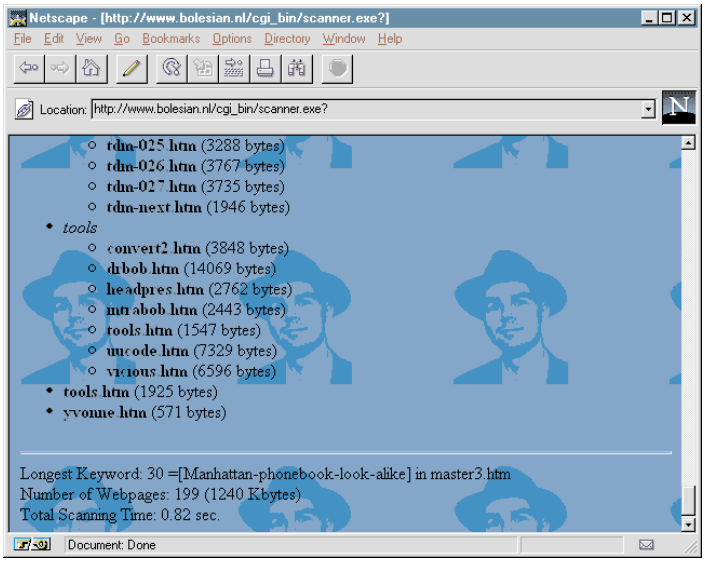
One of the first things we must do to write a static website search engine is analyse the website. How many sub-webpages are involved? How big is the longest (key)word? And so on.

Let's first decide what is a keyword and what is not, because that can have a big impact on the size of our index. For now, I've defined a keyword as starting with an alphabetic character and also optionally containing numeric characters and the + or – characters. This means that `iTec` is a valid keyword and so are `UK-BUG`, `IE4` and `C++`, but `Dr.Bob` fails (I can live with that, as the search engine is already on my website!). So here's the definitions:

```
const
  IdentSet = ['A'..'Z',
    'a'..'z', '0'..'9',
    '-', '+'];
  StartSet = ['A'..'Z',
    'a'..'z'];
```

With these two sets we can write a scanning application, upload it to

➤ *Figure 1*



the cgi-bin directory of our website (this assumes you indeed have a cgi-bin directory on your web server with execution capabilities, by the way). Assuming that the scanning application (let's call it "scanner" from now on) is executed from the cgi-bin directory, we need to do a `ChDir` to the root directory of the website (maybe one directory higher or somewhere else). Now that we're in the root of the website, we can recursively search for all the files in the current and sub-directories (remember that the scanner is a just another local application on the web server, so we don't need to use any difficult internet protocols here, just file operations). Using a `FindFirst ... FindNext ... FindClose` loop we can search for all files on the website. If a file has the .HTM or .ASP (Microsoft active server page) extension I assume it's a valid file to be indexed (you generally don't want to index binary files), so these files are opened and parsed for keywords.

There are a few special cases that we must exclude. First of all, some webmasters maintain their websites using Microsoft FrontPage. While I won't say anything bad about this tool, FrontPage has the habit of generating extra directories (starting with an underscore, such as _VTI_CNF) and files with FrontPage-specific maintenance and configuration information. These are not the kind of files you'd want to include in your index. Nor are the files with the .BAK extension (and sometimes the backup of HOME.HTM gets written to HOME.HTM.BAK), so we can only accept a file with .HTM or .ASP in its name if there isn't a .BAK in there as well.

Keeping these restrictions in mind, we can be relatively sure not to scan any unwelcome webpages (at least until someone finds another set of seemingly valid pages that need to be excluded as well: let me know if you find some).

The code for SCANNER.DPR (see Listing 1) keeps track of the number of webpages that are opened and the length of the longest keyword. Note that I ignore anything between < and > characters, so HTML tags and their (internal) contents are effectively ignored here. I decided not to base my website indexing on HTML Meta-Tags, but you could easily adapt these techniques to do that if you prefer.

After processing, these values are printed in an HTML output, so we can execute SCANNER.EXE from our cgi-bin directory and view the resulting HTML in a web browser. Figure 1 shows sample output.

Note that it takes only 0.82 seconds to scan my entire website of 199 pages. Of course, during internet peak hours (when the web

➤ *Listing 1*

```
{$APPTYPE CONSOLE}
{$I-}
uses
  SysUtils;
const
  website = 'http://www.drbob42.com';
  IdentSet = ['A'..'Z','a'..'z','0'..'9','-','+'];
  StartSet = ['A'..'Z','a'..'z'];
var
  f: Text;
  MaxFileName, MaxKeyword, Str: ShortString;
  { absolute = shortstring "length" hack }
  MaxLen: Byte absolute MaxKeyword;
  Len: Byte absolute Str;
  WebPages: Word = 0;
  Size: LongInt = 0;
procedure ScanFiles;
var
  SRec: TSearchRec;
  NotInTag: Boolean;
begin
  if FindFirst('*.*', faDirectory, SRec) = 0 then
    repeat
      if (SRec.Attr AND faDirectory) = faDirectory then
        begin
        if (SRec.Name[1] <> '.') then
          { skip '.' and '..' }
          if Pos('_vti',SRec.Name) = 0 then begin
            { _vti_cnf etc. }
            ChDir(SRec.Name);
            if IOResult = 0 then begin
              writeln('<LI><I>',SRec.Name,'</I>');
              writeln('<UL>');
              ScanFiles; { recursive!! }
              writeln('</UL>');
              ChDir('..')
            end else
              writeln('<LI><I>',SRec.Name,'</I> - locked')
          end
      end else if ((Pos('.HTM',UpperCase(SRec.Name)) > 0) or
        (Pos('.ASP',UpperCase(SRec.Name)) > 0)) and
        (Pos('.bak',SRec.Name) = 0)  then begin
        { file }
        writeln('<LI><B>',SRec.Name,
          '</B> (',SRec.Size,' bytes)');
        Size := Size + SRec.Size;
        assign(f,SRec.Name);
        reset(f);
        if IOResult = 0 then begin
          Inc(WebPages);
          NotInTag := True;
          while not eof(f) do begin
            Len := 0;
            while not eoln(f) do begin
              Inc(Len);
              read(f,Str[Len]);
              if not (Str[Len] in IdentSet) then begin
                Dec(Len);
                if (Len > MaxLen) and NotInTag then begin
                  MaxKeyword := Str;
                  MaxFileName := SRec.Name
                end;
                if Str[Len+1] = '>' then
                  NotInTag := True
                else
                  if Str[Len+1] = '<' then
                    NotInTag := False;
                Len := 0
              end else
                if (Len = 1) then
                  { start with letter ?? }
                  if not (Str[1] in StartSet) then
                    Len := 0
            end;
            if (Len > MaxLen) and NotInTag then begin
              MaxKeyword := Str;
              MaxFileName := SRec.Name
            end;
            readln(f)
          end;
          close(f)
        end
      end
    until FindNext(SRec) <> 0;
    FindClose(SRec)
  end {ScanFiles};
begin
  ChDir('..'); { get out of cgi-bin }
  if IOResult <> 0 then { skip };
  writeln('content-type: text/html');
  writeln;
  writeln('<HTML>');
  writeln('<BODY BACKGROUND="/gif/back.gif">');
  writeln('<H2>IndexBob</H2>');
  writeln('Scanning website ',website);
  writeln('<P>');
  writeln('<UL>');
  ScanFiles;
  writeln('</UL>');
  writeln('<HR>');
  writeln('Longest Keyword: ', MaxLen,
    ' =[',MaxKeyword,'] in ',MaxFileName);
  writeln('<BR>Number of Webpages: ',WebPages,' (',
    Size div 1024,' Kbytes)');
  writeln('<HR>');
  writeln('</BODY>');
  writeln('</HTML>')
end.
```

server is getting more simultaneous requests) performance might be lower.

Analysis Results

The results of running the Scanner CGI application on my website is as follows: 199 webpages, using 1240 Kb, with the longest keyword being 30 characters. Based on that information, I can define a structure for the webpage URLs and keywords we're about to index.

First of all, if I limit the number of webpages to 255, I can keep their number in a set, and store their names (the actual URLs) somewhere else. This means I can use one set of `1..255` to store the webpages where a specific keyword occurs, which means it takes 255 bits, that is 32 bytes, to store the webpage/keyword information for each keyword.

The maximum keyword length is currently 30 characters. However, this may change in the future, so I define a keyword as a `String[31]` (one character extra), which also takes 32 bytes (remember the length byte).

Combined, this means I need 64 bytes for each keyword to store the entire keyword information (keyword name with webpage bitset) and an additional string for each webpage or URL (up to 255 extra strings, as we noted earlier). See the declarations in Listing 2.

The entire structure of all the `TNode` records needs to be built while parsing the webpages in order to build the index. This structure must have a quick search facility, since we need to locate an existing keyword to add a webpage "bit" to the set of webpages where that keyword occurs, or insert a new keyword information record. So, the entire structure should at least be sorted and preferably use a fast algorithm for searching. I've decided to use a simple binary tree approach, which means that if I store `N` keywords I will only need `O(log N)` comparisons for each new keyword or update. This approach still means I can keep the application small and yet fast.

First, we need to define the `TTree` class itself (Listing 2). The `Keyword` data is stored in a field of type `TNode` and other than that we need two additional pointers to a sub-tree. We can define the tree to be alphabetically sorted on `Node.Keyword`, so the `Prev` sub-tree will hold all keywords "before" the current one and the `Next` sub-tree will hold all keywords "after" the current one.

The constructor is called from within a routine that found a keyword inside a webpage. So, while constructing the a new tree-item (a new *leaf*) we can immediately assign the keyword and initial webpage where we found this keyword. I also increase a global `Keywords` counter to keep track of the number of keywords that are found and the corresponding number of tree items that are created. Note that the `Prev` and `Next` sub-trees are set to `nil`, so this constructor effectively adds a "leaf" to a tree (with the potential to grow).

The use of this `Create` constructor (Listing 2) is embedded in the function `AddKeyword`, which in its turn is being called by the routine that's parsing the current webpage. `AddKeyword` gets the current (found) keyword as an argument, as well as the number of the current webpage which is being parsed.

A global variable `root` contains the "root" of the `TTree` of keywords. If no keywords have been found the `root` variable is `nil`. So `AddKeywords` checks the value of `root` and if it's `nil` we just create a new `TTree` and place it in the new `root`. If the `root` exists, we compare the new `Keyword` to the `root.Node.Keyword`. If the `Keyword` argument is "bigger" than (ie alphabetically comes after) the `root.Node.Keyword` we try the same comparison with the `Prev` sub-tree. If the `Keyword` argument is "smaller" we try the same comparison with the `Next` sub-tree and repeat this until we either find an end-leaf (ie no more subnodes) or a match for the keyword.

If the keyword isn't already in the tree we create a new node, passing the keyword and webpage ID to the constructor. Otherwise, we only need to add the webpage ID to the set of URLs (see `AddKeyword` in Listing 2).

➤ *Facing page: Listing 2*

The destructor's only purpose is to make sure all the nodes of the tree are destroyed and it's fairly simple: each node should first destroy the `Prev` subtree, followed by the `Next` subtree, followed by the destruction of the node itself.

When writing the binary tree to disk, it's very important to keep the sorted order, so we can read the binary tree in sorted order again (for the IndexBob main search engine application). See `WriteTree` in Listing 2.

WebSite Indexing

If we connect the scanner file parsing algorithm with the `AddKeyword` routine we end up with a binary tree of keywords. A binary tree is most efficient if each subtree has the same depth: a balanced binary tree. Of course, reading a number of webpages and creating nodes in the places where the new keywords belong doesn't guarantee a balanced binary tree, but rather a random binary tree. Fortunately, the tree will never be truly deep (unless we read a webpage where all the words are sorted to begin with) and the tree is only constructed once (during indexing) and will be written out to disk as soon as we're done.

With so much functionality in the entire `Index` unit (Listing 2), all that the main indexing program needs to do is declare itself a `CONSOLE` type application, and include the `Index` unit in the `uses` clause as follows:

```
{$APPTYPE CONSOLE}
uses Index;
end.
```

The index application itself writes the result with HTML tags to the standard output, so like SCANNER.EXE we can run this program (INDEX.EXE) from the cgi-bin directory, just like another standard CGI application. Where the scanner took less than a second to execute and return the HTML result, the indexer takes between 1 and 3 seconds to index my entire

```
unit Index;
{$I-}
interface
const
  website = 'http://www.drbob42.com';
  IdentSet = ['A'..'Z','a'..'z','0'..'9','-','+'];
  StartSet = ['A'..'Z','a'..'z'];
  MaxPage = 255;
type
  TNumPage = 0..MaxPage; { max number of webpages in site }
  TURLPage = ShortString { assuming URL <= 255 characters };
var WebPages: TNumPage = 0;
    WebPage: Array[TNumPage] of TURLPage;
const MaxKeyword = 31;
type
  TKeyword = String[MaxKeyword];
  TPageSet = Set of TNumPage;
  TNode = record
    Keyword: TKeyword; { 32 bytes }
    URLs: TPageSet;    { 32 bytes }
  end {TNode};
  TTree = class
    Node: TNode;
    constructor Create(const Keyword: TKeyword;
      WebPage: TNumPage);
    destructor Destroy; override;
  private
    Prev,Next: TTree;
  end {TTree};
var
  Keywords: Integer = 0;
  root: TTree = nil;
type
  TIndexFile = File of TNode;
implementation
uses SysUtils;
constructor TTree.Create(const Keyword: TKeyword;
  WebPage: TNumPage);
begin
  inherited Create;
  Inc(Keywords); // keep track of number of keywords
  Prev := nil;
  Next := nil;
  Node.Keyword := Keyword;
  Node.URLs := [WebPage]
end {Create};
destructor TTree.Destroy;
begin
  if Prev <> nil then Prev.Free;
  if Next <> nil then Next.Free;
  inherited Destroy
end {Destroy};
procedure AddKeyword(const Keyword: TKeyword;
  WebPage: TNumPage);
var tmp: TTree;
begin
  if root = nil then
    root := TTree.Create(Keyword,WebPage)
  else begin
    { search }
    tmp := root;
    repeat
      if tmp.Node.Keyword > Keyword then begin
        if tmp.Prev = nil then
          tmp.Prev := TTree.Create(Keyword,WebPage);
        tmp := tmp.Prev
      end else
        if tmp.Node.Keyword < Keyword then begin
          if tmp.Next = nil then
            tmp.Next := TTree.Create(Keyword,WebPage);
          tmp := tmp.Next
        end
    until tmp.Node.Keyword = Keyword;
    tmp.Node.URLs := tmp.Node.URLs + [WebPage]
  end
end {AddKeyword};
procedure ScanPage(const FileName: ShortString;
  WebPage: TNumPage);
var
  f: Text;
  NotInTag: Boolean;
  Keyword: ShortString;
  Len: Byte absolute Keyword; {absolute = shortstring "length" hack}
begin
  assign(f,FileName);
  reset(f);
  if IOResult = 0 then begin
    writeln('<LI><B>',FileName,'</B>');
    NotInTag := True;
    while not eof(f) do begin
      Len := 0;
      while not eoln(f) do begin
        Inc(Len);
        read(f,Keyword[Len]);
        if not (Keyword[Len] in IdentSet) then begin
          Dec(Len);
          if (Len > 2) and NotInTag then
            AddKeyword(LowerCase(Keyword),WebPage);
          if Keyword[Len+1] = '>' then
            NotInTag := True
          else
            if Keyword[Len+1] = '<' then
              NotInTag := False;
          Len := 0
        end else
          if (Len = 1) then
            { start with letter ?? }
            if not (Keyword[1] in StartSet) then
              Len := 0
      end;
      if (Len > 2) and NotInTag then
        AddKeyword(LowerCase(Keyword),WebPage);
      readln(f)
    end;
    close(f)
  end else
    writeln('<LI>',FileName); { failed to open }
end {ScanPage};
procedure ScanPages(const Path: ShortString);
var SRec: TSearchRec;
begin
  if FindFirst('*.*', faDirectory, SRec) = 0 then
    repeat
      if (SRec.Attr AND faDirectory) = faDirectory then
        begin
        if (SRec.Name[1] <> '.') then { skip '.' and '..' }
          if Pos('_vti',SRec.Name) = 0 then begin
            { _vti_cnf etc. }
            ChDir(SRec.Name);
            if IOResult = 0 then begin
              writeln('<LI><I>',SRec.Name,'</I>');
              writeln('<UL>');
              ScanPages(Path+'/'+SRec.Name); { recursive!! }
              writeln('</UL>');
              ChDir('..')
            end else
              writeln('<LI><I>',SRec.Name,'</I> - locked')
          end
        end else
          { file }
          if ((Pos('.HTM',UpperCase(SRec.Name)) > 0) or
            (Pos('.ASP',UpperCase(SRec.Name)) > 0)) and
            (Pos('.bak',SRec.Name) = 0)  then begin
            WebPage[WebPages] := Path + '/' + SRec.Name;
            ScanPage(SRec.Name,WebPages);
            Inc(WebPages)
          end
    until FindNext(SRec) <> 0;
    FindClose(SRec)
end {ScanPages};
procedure WriteTree(var IndexFile: TIndexFile; root: TTree);
begin
  if root.Prev <> nil then
    WriteTree(IndexFile,root.Prev);
  write(IndexFile,root.Node);
  if root.Next <> nil then
    WriteTree(IndexFile,root.Next);
end {WriteTree};
var
  i: Integer;
  PageFile: Text;
  IndexFile: TIndexFile;
initialization
  ChDir('..');
  if IOResult <> 0 then { skip };
  writeln('content-type: text/html');
  writeln;
  writeln('<HTML>');
  writeln('<BODY BACKGROUND="/gif/back.gif">');
  writeln('<H2>IndexBob</H2>');
  writeln('Creating index for: ',website);
  writeln('<P>');
  writeln('<UL>');
  ScanPages(website);
  writeln('</UL>');
  ChDir('cgi-bin');
  if IOResult <> 0 then { skip };
  assign(PageFile,'pages.bob');
  try
    rewrite(PageFile);
    for i:=0 to WebPages-1 do
      writeln(PageFile,WebPage[i]);
  finally
    close(PageFile)
  end;
  assign(IndexFile,'index.bob');
  if root <> nil then
    try
      rewrite(IndexFile);
      WriteTree(IndexFile,root)
    finally
      close(IndexFile)
    end;
  writeln('<HR>');
  writeln('<FONT SIZE=1>');
  writeln('Webpages: ',WebPages);
  writeln('<BR>Keywords: ',Keywords);
  writeln('</FONT>');
  writeln('<HR>');
  writeln('</BODY>');
  writeln('</HTML>')
finalization
  root.Free
end.
```

website with 199 pages (and over 8,000 keywords) and return the HTML output. The filesize of the index is 8,000+ times 64 bytes, which is slightly under 0.5Mb. However, since we accepted any keyword with three or more characters, we also accepted dozens (if not hundreds) of keywords like `the`, `and` and `you` which may occur in every webpage and are not very useful when building a keyword index. We'll keep that in mind as a possible enhancement for next month...

Figure 2 shows the output from a slightly more advanced version of the Index program, but it is a "live" screenshot from our own company intranet server.

So, for a website with up to 255 pages, we've just seen two CGI applications that can help us to produce an index of keywords and URL webpage names. The scanner first counts the .HTM and .ASP webpages as well as the length of the longest keyword, and the indexer then creates an internal binary tree with the keyword information and set of webpage IDs and writes it to disk as a sorted file of `TNode`.

The next step is the real search engine itself: another CGI application that accepts a keyword, loads the keyword index file in a balanced binary tree, finds the keyword in the tree (or not as the case may be) and produces a list of webpage names from the set of URL IDs and the file of webpage names The remainder of this article will show a very basic first approach to this search engine. Next time we'll explore a number of additional features and enhancements.

Dr.Bob's CGI

First of all, we need a supporting unit (small and fast if possible) to handle CGI requests. In my previous internet-related *Under Construction* columns, I played with a `TDosEnvironment` component that we could create dynamically to read DOS environment variables etc. However, every CGI application that I wrote would start out with a copy of the very first (debug) CGI application, as so much code was ready to be used again. And while I was using components, I was making bad use of re-use techniques, so I decided to put all the CGI techniques in one unit with an interface to the obtained CGI query values. The end result is the `DrBobCGI` unit, which can be put in the `uses` clause of any CGI application and will shield us from any further low-level CGI details from now on. All we need to do is call the `Values` function with the name of the input field we're looking for as the argument (`Keyword` in our case):

```
function Value(const Field:
  ShortString): ShortString;
```

This single function is actually all we need to communicate with the CGI query. The source code for the `DrBobCGI` unit is on this month's disk. Note that the `ContentLength` is also defined in the interface of `DrBobCGI`, so we can examine this and decide not to bother calling `Value` if `ContentLength` is 0.

IndexBob

Apart from the `Value` function from `DrBobCGI` we also need to create a balanced binary tree, load the sorted indexfile of keywords, the file of URL webpage names, obtain the required keyword (using the `Value` function from `DrBobCGI`), search for the keyword in the binary tree and present the associated URLs to the user.

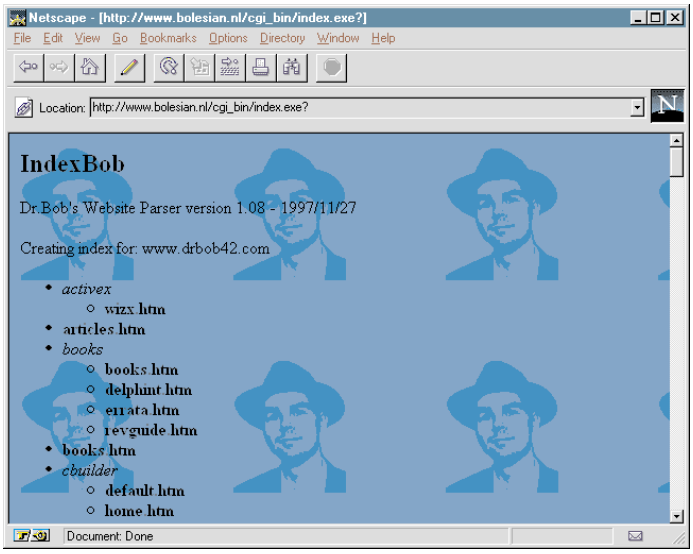
The interface part of the `IndexBob` unit is almost the same as the interface part of the `Index` unit, with the exception of the `TTree` class, which is slightly different: in `Index` the binary tree is created dynamically, in `IndexBob` the tree can be created as a balanced binary tree and loaded from disk.

We can obtain the number of keywords by looking at the filesize of INDEX.BOB. Using the number of keywords, we can call `CreateRoot` to create a full tree for the highest power of 2 minus 1 that's just smaller than this number of keywords (so for 100 keywords, we first create a tree with 1+2+4+8+16+32 = 63 nodes, for example). After that first "complete" binary tree, we need to add a number of "extra" nodes, namely the number of keywords minus the nodes already created (in the above example, that's 100 minus 63 = 37 extra nodes) using a call to `CreateLeafs` with the number of extra nodes to create (Listing 3).

Once we've created a balanced binary tree (with the same number of nodes as records in the file of nodes) we can "walk" the tree in order and read the nodes (which are also in alphabetically sorted order on the disk) as shown in `ReadNode` in Listing 3.

The unit does all the work, all the main CGI program IndexBob needs to do is include the unit `IndexBob` (along with `DrBobCGI`) and use the `Keyword` (calling `Values` from `DrBobCGI`) to search for a `Found` set of URLs (calling `FindKeywordInPages` from `IndexBob`) and presenting the URLs as an HTML formatted list.

➤ *Figure 2*



```
unit IndexBob;
{$I-}
interface
{ ** CODE OMITTED: SEE DISK FOR FULL LISTING ** }
type
  { ** CODE OMITTED: SEE DISK FOR FULL LISTING ** }
  TTree = class
    Node: TNode;
    constructor Create;
    destructor Destroy; override;
    function FindKeywordInPages(const KeyWord: TKeyword):
      TPageSet;
  private
    Prev,Next: TTree;
  end {TTree};
{ ** CODE OMITTED: SEE DISK FOR FULL LISTING ** }
implementation
function Pages(PageSet: TPageSet): Byte;
  { return the number of "1" (set) bits in the set, which
    does mean: the number of URLs containing the keyword }
var B: Byte;
begin
  Result := 0;
  for B := 0 to MaxPage do
    if B in PageSet then Result := Result + 1
end {Pages};
{ ** CODE OMITTED: SEE DISK FOR FULL LISTING ** }
function TTree.FindKeywordInPages(const Keyword: TKeyword):
  TPageSet;
var tmp: TTree;
begin
  Result := []; { assume keyword isn't found }
  tmp := root;
  repeat
    if tmp.Node.Keyword > Keyword then
      tmp := tmp.Prev
    else
      if tmp.Node.Keyword < Keyword then
        tmp := tmp.Next
  until (tmp = nil) or (tmp.Node.Keyword = Keyword);
  if tmp <> nil then
    Result := tmp.Node.URLs  { keyword is found }
end {FindKeywordInPages};
function CreateRoot(depth: Integer): TTree;
var r: TTree;
begin
  if depth > 0 then begin
    r := TTree.Create;
    r.Prev := CreateRoot(depth-1);
    r.Next := CreateRoot(depth-1);
    CreateRoot := r
  end else
    CreateRoot := nil
end {CreateRoot};
procedure CreateLeafs(var number: Integer; root: TTree);
begin
  if root.Prev <> nil then begin
    CreateLeafs(number,root.Prev);
    if number > 0 then
      CreateLeafs(number,root.Next)
  end else begin
    root.Prev := TTree.Create;
    Dec(number);
    if number > 0 then begin
      root.Next := TTree.Create;
      Dec(number)
    end
  end
end {CreateLeafs};
procedure ReadNode(var IndexFile: TIndexFile; root: TTree);
begin
  if root.Prev <> nil then
    ReadNode(IndexFile, root.Prev);
  read(IndexFile,root.Node);
  Inc(Keywords);
  if root.Next <> nil then
    ReadNode(IndexFile, root.Next)
end {ReadNode};
var
  PageFile: Text;
  IndexFile: TIndexFile;
  total,depth,i: Integer;
initialization
  writeln('content-type: text/html');
  writeln;
  writeln('<HTML>');
  writeln('<BODY BACKGROUND="/gif/back.gif">');
  writeln('<H2>IndexBob</H2>');
  { ** CODE OMITTED: SEE DISK FOR FULL LISTING ** }
  writeln('<P>');
  writeln('<FORM METHOD="POST" '+
    'ACTION="/cgi-bin/indexbob.exe">');
  writeln('<TABLE>');
  writeln('<TR><TD><I>Search again:</I></TD>');
  writeln('<TD><INPUT TYPE="TEXT" NAME="Keyword" '+
    'SIZE=29></TD></TR>');
  writeln('<TR><TD></TD><TD>');
  writeln('<INPUT TYPE="SUBMIT" VALUE="Search">');
  writeln('<INPUT TYPE="RESET" VALUE="Reset"></TD></TR>');
  writeln('</TABLE>');
  writeln('</FORM>');
  { ** CODE OMITTED: SEE DISK FOR FULL LISTING ** }
  writeln('<HR>');
  assign(PageFile,'pages.bob');
  reset(PageFile);
  if IOResult = 0 then begin
    while not eof(PageFile) do begin
      readln(PageFile,WebPage[WebPages]);
      Inc(WebPages)
    end;
    close(PageFile)
  end;
  assign(IndexFile,'index.bob');
  reset(IndexFile);
  total := FileSize(IndexFile);
  if IOResult = 0 then begin
    if total = 1 then
      root := TTree.Create
    else begin
      {total > 1}
      depth := 0;
      i := 1;
      repeat
        i := i SHL 1;
        Inc(depth)
      until i >= total;
      Dec(depth);
      i := total - (i SHR 1) + 1;
      root := CreateRoot(depth);
      if i > 0 then
      CreateLeafs(i, root)
    end;
    if total > 0 then
      ReadNode(IndexFile, root);
    close(IndexFile)
  end
finalization
  writeln('<HR>');
  writeln('<FONT SIZE=1>');
  writeln('Webpages: ',WebPages);
  writeln('<BR>Keywords: ',Keywords);
  writeln('</FONT>');
  writeln('<HR>');
  writeln('</BODY>');
  writeln('</HTML>');
  root.Free
end.
```

➤ *Listing 3*

See Listing 4. The webpages (URLs) that contain the keyword we're looking for are presented in a list, where we can click on the URL itself, by embedding them in a

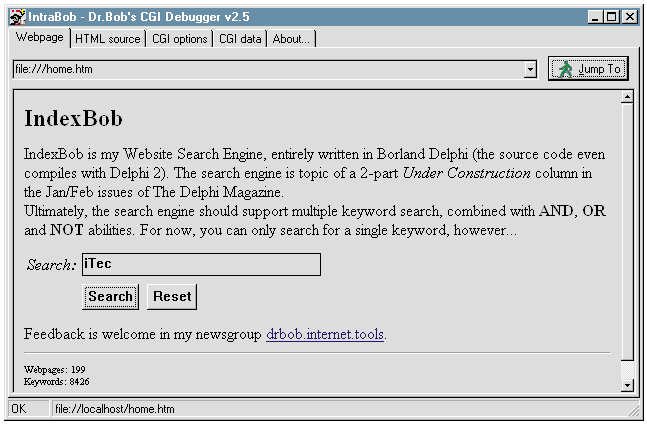
```
<A HREF="..."> ... </A>
```

hyperlink. The only thing we're missing here is some more information to accompany the URL, but that's also a topic we will return to next month...

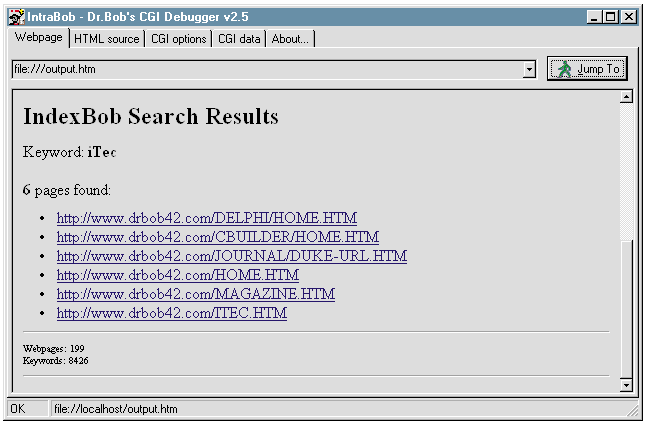
This shows how we can search for a single keyword in the website index and using an HTML CGI form and IntraBob, we can show how this would look like in real life. The HTML CGI form is called HOME.HTM (a less verbose version is included on the disk), see Listing 5.

Using IntraBob v2.5, we can see the mini website search engine in action (Figure 3). By the time you read this article you can also try the real thing on my website at www.drbob42.com. The result from the search for the keyword iTec can be seen in Figure 4: a total of 6 webpages are found.

If you try the final website search engine IndexBob (either by using IntraBob or by running it on a web server, such as from my website), you'll notice that the speed is very good. Loading the index and locating the keyword is done within 0.1 to 0.3 seconds and generally I get an answer back within one second. Sending that output webpage over the internet back to my browser takes more time than



➤ *Figure 3*



➤ *Figure 4*

```
{$APPTYPE CONSOLE}
uses
  SysUtils,
  DrBobCGI, // read Keyword from CGI "Form"
  IndexBob; // read index.bob and pages.bob
var
  Keyword: ShortString;
  Found: TPageSet;
  i: Integer;
begin
  writeln('<H2>IndexBob Search Results</H2>');
  Keyword := Value('Keyword');
  writeln('Keyword: <B>',Keyword,'</B><BR>');
  if root <> nil then
    if Length(Keyword) in [3..MaxKeyword] then
      Found := Found * root.FindKeywordInPages(LowerCase(Keyword))
    else
      Found := []; // no pages found
  writeln('<BR>');
  writeln('<B>',Pages(Found),'</B> pages found:');
  writeln('<UL>');
  for i:=0 to WebPages-1 do
    if i in Found then
      writeln('<LI><A HREF="',WebPage[i],'">',WebPage[i],'</A>');
  writeln('</UL>');
end.
```

➤ *Listing 4*

```
<HTML>
<BODY>
<H2>IndexBob</H2>
<FORM METHOD="POST" ACTION="/cgi-bin/indexbob.exe">
Search:
<INPUT TYPE="TEXT" NAME="Keyword" SIZE=29>
<INPUT TYPE="SUBMIT" VALUE="Search">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

➤ *Listing 5*

actually looking for the index in the indexfile. Of course, this may change when we start to look for more keywords, or combine queries, but we'll find out next time...

Future

Of course, IndexBob isn't done yet. As I've indicated, next month we will see how to extend and enhance IndexBob. Among the ideas for improvements are: searching for multiple keywords (using AND), searching for alternatives (using OR), searching using filters (using NOT), showing part of the content of the found URL (like the page Title) instead of just the URL itself and limiting the indexfile by filtering common words.

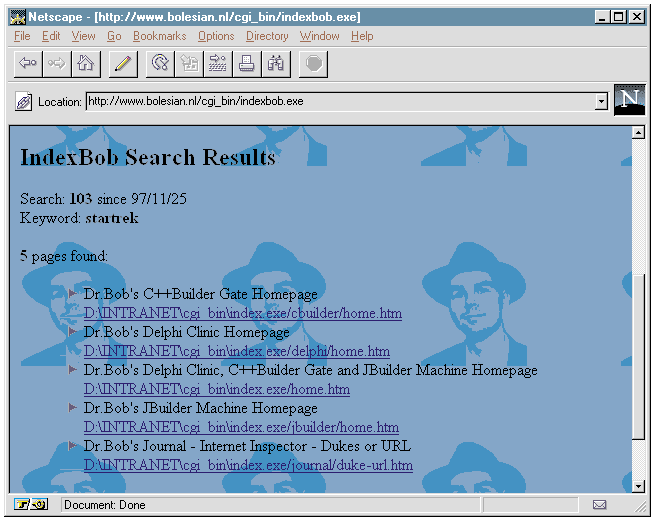
Finally, I've had some ideas to speed things up even further and to enable IndexBob to be used for websites with more than 255 webpages.

I'm also investigating the possibility of searching for only subparts of a website (for example only search for "database" in my "book reviews" section, or only search for "profiler" in my "articles" and "tips" sections).

One thing that I've added already is a search logging ability, so I can find out how other people are using the search engine on my website, to give me feedback on how to possibly improve both the search engine and the website itself (Figure 5).

If you have any other ideas for improvements, or just want to talk about IndexBob, don't hesitate to send me some feedback in my special newsgroup drbob.internet.tools at news.shoresoft.com (there's a link on my website). I welcome feedback and it can only help us all.

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a professional knowledge engineer technical consultant using Delphi, C++Builder and JBuilder for Bolesian (www.bolesian.com), a freelance technical author for *The Delphi Magazine* and co-author of *The Revolutionary Guide to Delphi 2*. Bob is now working on an electronic knowledge base called *Delphi Internet Solutions*, with topics about Delphi and the internet/intranet. In his spare time, Bob likes to watch video tapes of *Star Trek Voyager* and *Deep Space Nine* with his 3.5-year old son Erik Mark Pascal and his 1-year old daughter Natasha Louise Delphine.


Netscape - [http://www.bolesian.nl/cgi_bin/indexbob.exe]
File Edit View Go Bookmarks Options Directory Window Help
Location: http://www.bolesian.nl/cgi_bin/indexbob.exe
IndexBob Search Results
Search: 103 since 97/11/25
Keyword: startrek
5 pages found:
Dr.Bob's C++Builder Gate Homepage
D:\INTRANET\cgi_bin\index.exe/cbuilder/home.htm
Dr.Bob's Delphi Clinic Homepage
D:\INTRANET\cgi_bin\index.exe/delphi/home.htm
Dr.Bob's Delphi Clinic, C++Builder Gate and JBuilder Machine Homepage
D:\INTRANET\cgi_bin\index.exe/home.htm
Dr.Bob's JBuilder Machine Homepage
D:\INTRANET\cgi_bin\index.exe/jbuilder/home.htm
Dr.Bob's Journal - Internet Inspector - Dukes or URL
D:\INTRANET\cgi_bin\index.exe/journal/duke-url.htm
Document: Done

➤ *Figure 5*