1. Shin Sort and Search Program


```
procedure Sort_Search_Keys;

{Print_Sorted_Keys will print out stored keys in Shin tree traversing preorder
 tree traversal.  While traversing, it stores tree node's character in stack for
 printing. If it hits leaf node, it prints out characters for stored key reading
 from stack and pops up node from the stack if the portion is not necessary to
 be kept for other not yet printed key any further. It pushes and pops node by
 changing top of the stack until it finishes its traversing.}
Procedure Print_Sorted_Keys (Tree_Pt: Shin_Node_Pt);
  var
    Stack_Item_Pt : Shin_Node_Pt;
    Counter : integer;
    Second_Stack_Top, Second_Stack_Pt: Shin_Node_Pt;

  procedure Push_Node_Into_Stack;
  begin
    New(Stack_Item_Pt);
    Stack_Item_Pt^.Node_Char := Tree_Pt^.Node_Char;
    if Sort_Debug then
      begin
        write(Sort_Out_File,'', Stack_Item_Pt^.Node_Char, ''' pushed --> ');
        Push_Pop_Counter := Push_Pop_Counter + 1;
        if Push_Pop_Counter mod COUNT_PUSH_POP = 0 then
          begin
            writeln(Sort_Out_File);
            Push_Pop_Counter := 0;
          end;
      end;
    Stack_Item_Pt^.Left_Son := Stack_Top;
    Stack_Top := Stack_Item_Pt;
  end;
```

```
procedure Pop_Up_Node_From_Stack;
begin
  if Stack_Top <> nil then
    begin
      if Sort_Debug then
        begin
          write(Sort_Out_File,"", Stack_Top^.Node_Char,'" popped --> ');
          Push_Pop_Counter := Push_Pop_Counter + 1;
          if Push_Pop_Counter mod COUNT_PUSH_POP = 0 then
            begin
              writeln(Sort_Out_File);
              Push_Pop_Counter := 0;
            end;
        end;
      Stack_Top := Stack_Top^.Left_Son;
    end;
end;

procedure Print_Out_Key;
var
  i : integer;
begin
  Second_Stack_Top := nil;
  pt := Stack_Top;
  while pt <> nil do
    begin
      New(Second_Stack_Pt);
      Second_Stack_Pt^.Node_Char := pt^.Node_Char;
      second_Stack_Pt^.Left_Son := Second_Stack_Top;
      Second_Stack_Top := Second_Stack_Pt;
      pt := pt^.Left_Son;
    end;
  No_Out_Keys := No_Out_Keys + 1;
  i := 0;
  if Push_Pop_Counter mod COUNT_PUSH_POP <> 0 then
```

```
    writeln(Sort_Out_File);
  write(Sort_Out_File,'      The stored key, "');
  pt := Second_Stack_Top;
  while pt <> nil do
    begin
      i := i + 1;
      write(Sort_Out_File, pt^.Node_Char);
      Output_List[No_Out_Keys].Key_Arr[i] := pt^.Node_Char;
      pt := pt^.Left_Son;
    end;
 writeln(Sort_Out_File, '", is printed out here.');
 Push_Pop_Counter := 0;
 Output_List[No_Out_Keys].Char_Counter := i;
end;

procedure Print_Out_Key_Based_On_Counter;
var
  i : integer;
begin
    Counter := Tree_Pt^.Key_Counter;
    while Counter > 0 do
      begin
        Second_Stack_Top := nil;
        pt := Stack_Top;
        while pt <> nil do
          begin
            New(Second_Stack_Pt);
            Second_Stack_Pt^.Node_Char := pt^.Node_Char;
            second_Stack_Pt^.Left_Son := Second_Stack_Top;
            Second_Stack_Top := Second_Stack_Pt;
            pt := pt^.Left_Son;
          end;
        No_Out_Keys := No_Out_Keys + 1;
        i := 0;
        if Push_Pop_Counter mod COUNT_PUSH_POP <> 0 then
          writeln(Sort_Out_File);
```

```
            write(Sort_Out_File,'      The stored key, "');
            pt := Second_Stack_Top;
            while pt <> nil do
              begin
                i := i + 1;
                write(Sort_Out_File, pt^.Node_Char);
                Output_List[No_Out_Keys].Key_Arr[i] := pt^.Node_Char;
                pt := pt^.Left_Son;
              end;
            writeln(Sort_Out_File, '", is printed out here.');
            Push_Pop_Counter := 0;
            Output_List[No_Out_Keys].Char_Counter := i;
            Counter := Counter - 1;
          end;
  end;

  begin      {---------- Print_Sorted_Keys Start from here ----------}
      if Tree_Pt <> nil then
        begin
          Push_Node_Into_Stack;     {Store the current node}
          if Tree_Pt^.Left_Son = nil then
            Print_Out_Key;
          if Tree_Pt^.Key_Counter > 0 then
            Print_Out_Key_Based_On_Counter;   {Print out more as counted}
          Print_Sorted_Keys (Tree_Pt^.Left_Son);
          Pop_Up_Node_From_Stack;     {Only one Shin's node popped up}
          Print_Sorted_Keys(Tree_Pt^.Right_Son);
        end;
  end;       {End of Procedure Print_Sorted_Keys}


{Shin_Sort recursive routine searches right position to insert input node in
 the Shin tree by comparing input key node's character with tree's node
 character.}
  procedure Shin_Sort(Shin_Tree_Ptr: Shin_Node_Pt; Key_Ptr: Shin_Node_Pt);
    begin
```

```
if Shin_Tree_Ptr = nil then
   case Header_LR_Flag of
      left : Header_Ptr^.Left_Son := Key_Ptr;
      right: Header_Ptr^.Right_Son:= Key_Ptr;
   end
else
   if ord(Key_Ptr^.Node_Char) = ord(Shin_Tree_Ptr^.Node_Char) then
   begin
      Header_Ptr := Shin_Tree_Ptr;
      Header_LR_Flag := left;
      Shin_Tree_Ptr := Shin_Tree_Ptr^.Left_Son;
      if (Shin_Tree_Ptr = nil) or (Key_Ptr^.Left_Son = nil) then
         begin
            Header_Ptr^.Key_Counter := Header_Ptr^.Key_Counter + 1;
            if Shin_Tree_Ptr = nil then
               begin
                  Key_Ptr := Key_Ptr^.Left_Son;
                  Header_Ptr^.Left_Son := Key_Ptr;
               end
         end
      else
         begin
            Key_Ptr := Key_Ptr^.Left_Son;
            Shin_Sort(Shin_Tree_Ptr, Key_Ptr);
         end;
   end

else if ord(Key_Ptr^.Node_Char) < ord(Shin_Tree_Ptr^.Node_Char) then
   begin
      case Header_LR_Flag of
         left : Header_Ptr^.Left_Son := Key_Ptr;
         right: Header_Ptr^.Right_Son:= Key_Ptr;
      end;
      Key_Ptr^.Right_Son := Shin_Tree_Ptr;
   end
```

```
    else if ord(Key_Ptr^.Node_Char) > ord(Shin_Tree_Ptr^.Node_Char) then
      begin
        Header_Ptr := Shin_Tree_Ptr;
        Header_LR_Flag := right;
        Shin_Tree_Ptr := Shin_Tree_Ptr^.Right_Son;
        Shin_Sort(Shin_Tree_Ptr, Key_Ptr);
      end;

  end;          {End of Shin_Sort}

{Shin_Sort_The_Key calls Shin_sort recursive routine comparing Shin tree's node's
 character value with input string's pointed character}
  procedure Shin_Sort_The_Key(var Root_Ptr: Shin_Node_Pt; Key_Ptr: Shin_Node_Pt);
  begin

    if (Sort_Debug and Accurate_Debug) and
       ((Root_Ptr <> nil) and (Key_Ptr <> nil))then
      writeln(Sort_Out_File, 'Root Char: ',Root_Ptr^.Node_Char,'     vs     ',
                                    'Key Char: ',Key_Ptr^.Node_Char);
    if(Key_Ptr = nil) then
      writeln(Sort_Out_File, 'Sort Error -- The key input is empty.');
    if Root_Ptr = nil then
      Root_Ptr := Key_Ptr

    else if ord(Root_Ptr^.Node_Char) > ord(Key_Ptr^.Node_Char) then
      begin
        Key_Ptr^.Right_Son := Root_Ptr;
        Root_Ptr := Key_Ptr;
      end

    else if ord(Root_Ptr^.Node_Char) <= ord(Key_Ptr^.Node_Char) then
      begin
        Shin_Tree_Pointer := Root_Ptr;
        Shin_Sort(Shin_Tree_Pointer, Key_Ptr);
      end;
  end;          {End of Shin_Sort_The_Key}
```

```
{Link-up the input key characters with their node's left child pointer.}
 procedure Read_and_Store_Key(var R_Pt: Shin_Node_Pt);
    var
      i : integer;
      Shin_Pt : Shin_Node_Pt;
    begin
      Str_Size := 0;
      readln(Sort_Keys_File, Key_String);
      Str_Size := Length(Key_String);
      if Sort_Debug then writeln(Sort_Out_File, 'Preorder after key #',No_Keys:1,
        ', "', Key_String,'" is inserted.');
      if Str_Size <= 0 then
        R_Pt := nil
      else
        begin
          New(Shin_Pt);
          R_Pt := Shin_Pt;
          Shin_Pt^.Node_Char := Key_String[1];
          Shin_Pt^.Key_Counter := 0;
          Shin_Pt^.Left_Son := nil;
          Shin_Pt^.Right_Son := nil;
          pt := R_Pt;
          if Str_Size >= 2 then
            for i := 2 to Str_Size do
              begin
                New(Shin_Pt);
                pt^.Left_Son := Shin_Pt;
                Shin_Pt^.Node_Char := Key_String[i];
                Shin_Pt^.Key_Counter := 0;
                Shin_Pt^.Left_Son := nil;
                Shin_Pt^.Right_Son := nil;
                pt := Shin_Pt;
              end;
          if Sort_Debug then
```

```
            begin
              if Accurate_Debug then write(Sort_Out_File, 'Linked Input Key String: ');
              pt := R_Pt;
              for i := 1 to Str_Size do
                if pt <> nil then
                  begin
                    if Accurate_Debug then write(Sort_Out_File, pt^.Node_Char);
                    Input_List[No_Keys].Key_Arr[i] := pt^.Node_Char;
                    pt := pt^.Left_Son;
                  end;
              Input_List[No_Keys].Char_Counter := Str_Size;
              if Accurate_Debug then writeln(Sort_Out_File);
            end;
        end;
    end;    {End of Read_And_Store_Key}


{Procedure Search_Key_In_Tree will read a file for keys to be searched in
 Shin tree.  It performs searching by calling Shin_Search recursive module
 for each key.}
  procedure Search_Key_In_Tree(Key_Pointer: Shin_Node_Pt);
  var
    Counter_Flag : integer;

  procedure Key_Found;
  begin
    writeln(Sort_Out_File, '       The key, ''', Key_String,
                           ''', is found in the Shin tree.');
  end;

  procedure Key_Not_Found;
  begin
    writeln(Sort_Out_File, '       The key, ''', Key_String, ''', is not found.');
  end;

  procedure Read_Key_For_Search(var R_Pt: Shin_Node_Pt);
```

```
var
  i : integer;
  Shin_Pt : Shin_Node_Pt;
begin
  Str_Size := 0;
  readln(Search_Key_File, Key_String);
  Str_Size := Length(Key_String);
  if Sort_Debug then
    begin
      writeln(Sort_Out_File);
      writeln(Sort_Out_File, Key_Num:3,'. Search the Key: ',
                                    Key_String,'      Size: ',Str_Size:1);
    end;
  if Str_Size <= 0 then
    R_Pt := nil
  else
  begin
    New(Shin_Pt);
    R_Pt := Shin_Pt;
    Shin_Pt^.Node_Char := Key_String[1];
    Shin_Pt^.Key_Counter := 0;
    Shin_Pt^.Left_Son := nil;
    Shin_Pt^.Right_Son := nil;
    pt := R_Pt;
    if Str_Size >= 2 then
      for i := 2 to Str_Size do
      begin
        New(Shin_Pt);
        pt^.Left_Son := Shin_Pt;
        Shin_Pt^.Node_Char := Key_String[i];
        Shin_Pt^.Key_Counter := 0;
        Shin_Pt^.Left_Son := nil;
        Shin_Pt^.Right_Son := nil;
        pt := Shin_Pt;
      end;
    if Sort_Debug and Accurate_Debug then
```

```
            begin
              write(Sort_Out_File, 'Linked Input Key String: ');
              pt := R_Pt;
              for i := 1 to Str_Size do
              if pt <> nil then
                begin
                  write(Sort_Out_File, pt^.Node_Char);
                  pt := pt^.Left_Son;
                end;
              writeln(Sort_Out_File);
            end;
        end;
    end;      {End of Read_Key_For_Search}

{Shin_Search recursive procedure tells whether input key is in the Shin tree
or not, by comparing input key node's character with current tree node's
character and moving closer in the tree.}
 procedure Shin_Search(Shin_Tree_Ptr: Shin_Node_Pt; Key_Ptr: Shin_Node_Pt);
 begin
    if (Shin_Tree_Ptr = nil) or (Key_Ptr = nil) then
      begin
        if (Shin_Tree_Ptr = nil) and (Key_Ptr = nil) then
          begin
            Key_Found
          end
        else if (Shin_Tree_Ptr = nil) and (Key_Ptr <> nil) then
          begin
            Key_Not_Found
          end
        else if (Shin_Tree_Ptr <> nil) and (Key_Ptr = nil) then
          begin
            if Counter_Flag > 0 then
              begin
                Key_Found
              end
            else
```

```
            begin
              Key_Not_Found;
            end
        end
    end
  else
    begin
      if Key_Ptr^.Node_Char = Shin_Tree_Ptr^.Node_Char then
        begin
          if Sort_Debug then
            begin
              writeln(Sort_Out_File, '        Character of the Node: ',
              Shin_Tree_Ptr^.Node_Char,'      Counter Value of the Node: ',
              Shin_Tree_Ptr^.Key_Counter:1);
            end;
           if Shin_Tree_Ptr^.Key_Counter > 0 then
             begin
               Counter_Flag := Shin_Tree_Ptr^.Key_Counter;
            end
          else
            begin
              Counter_Flag := 0;
            end;
          Key_Ptr := Key_Ptr^.Left_Son;
          Shin_Tree_Ptr := Shin_Tree_Ptr^.Left_Son;
          Shin_Search(Shin_Tree_Ptr, Key_Ptr);
        end
      else if Key_Ptr^.Node_Char > Shin_Tree_Ptr^.Node_Char then
        begin
          Shin_Tree_Ptr := Shin_Tree_Ptr^.Right_Son;
          Shin_Search(Shin_Tree_Ptr, Key_Ptr);
        end
      else if Key_Ptr^.Node_Char < Shin_Tree_Ptr^.Node_Char then
        Key_Not_Found;
    end;
end;    {End of Recursive Shin Search Routine}
```

```
  begin     {Procedure Search_Key_In_Tree starts from here}
    Counter_Flag := 0;
    Key_Num := 0;
    if Root_Pt = nil then writeln(' Tree is empy not to be searched!')
    else
      begin
        while not EOF(Search_Key_File) do
          begin
            Key_Num := Key_Num + 1;
            Read_Key_For_Search(Key_Pointer);
            Shin_Tree_Pointer := Root_Pt;
            Shin_Search (Shin_Tree_Pointer, Key_Pointer);
          end;
      end;
  end;      {End of Procedure Search_Key_In_Tree}

{Print_Input_List will print out the keys which are read for being sorted.}
  procedure Print_Input_List;
  var
    i, j, Line_Chars : integer;
  begin
    writeln(Sort_Out_File);
    writeln(Sort_Out_File);
    writeln(Sort_Out_File,'----------<Input List for the ',No_Keys:1,' Keys>----------');
    Line_Chars := 0;
    Hit_Margin_Flag := false;
    for i := 1 to No_Keys do
      begin
        j := 1;
        while j <= Input_List[i].Char_Counter do
          begin
            write(Sort_Out_File, Input_List[i].Key_Arr[j]);
            j := j + 1;
            Line_Chars := Line_Chars + 1;
          end;
```

```
            write(Sort_Out_File, '    ');
            Line_Chars := Line_Chars + 3;
            if Line_Chars >= LINE_MARGIN then
               Hit_Margin_Flag := true;
            if Hit_Margin_Flag then
               begin
                  writeln(Sort_Out_File);
                  Hit_Margin_Flag := false;
                  Line_Chars := 0;
               end;
         end;
   end;

{Print_Output_List will print out the keys that are sorted, so they will be printed in order.}
   procedure Print_Output_List;
   var
      i, j, Line_Chars: integer;
   begin
      writeln(Sort_Out_File);
      writeln(Sort_Out_File);
      writeln(Sort_Out_File,'----------<Sorted List for the ',No_Out_Keys:1,' Keys>----------');
      Line_Chars := 0;
      Hit_Margin_Flag := false;
      for i := 1 to No_Out_Keys do
         begin
            j := 1;
            while j <= Output_List[i].Char_Counter do
               begin
                  write(Sort_Out_File, Output_List[i].Key_Arr[j]);
                  j := j + 1;
                  Line_Chars := Line_Chars + 1;
               end;
            write(Sort_Out_File, '    ');
            Line_Chars := Line_Chars + 3;
            if Line_Chars >= LINE_MARGIN then
               Hit_Margin_Flag := true;
```

```
        if Hit_Margin_Flag then
          begin
            writeln(Sort_Out_File);
            Hit_Margin_Flag := false;
            Line_Chars := 0;
          end;
      end;
  end;

{This recursive routine will traverse Shin tree in preorder.}
  procedure Preorder_Traversal(Shin_Node: Shin_Node_Pt);
  begin
    if Shin_Node <> nil then
      begin
        if Shin_Node^.Key_Counter > 0 then
          begin
            write(Sort_Out_File, Shin_Node^.Node_Char, ':',Shin_Node^.Key_Counter:1,' ');
            Order_Count := Order_Count + 4;
            Temp_Cnt := Order_Count mod PRINT_LINE_DIVISOR;
              if (Temp_Cnt >= 0) and (Temp_Cnt <= 3) then writeln(Sort_Out_File);
          end
        else
          begin
            write(Sort_Out_File, Shin_Node^.Node_Char,' ');
            Order_Count := Order_Count + 2;
            if ((Order_Count mod PRINT_LINE_DIVISOR) = 0) or
               ((Order_Count mod PRINT_LINE_DIVISOR) = 1) then
              writeln(Sort_Out_File);
          end;
        Preorder_Traversal(Shin_Node^.Left_Son);
        Preorder_Traversal(Shin_Node^.Right_Son);
      end;
  end;

  procedure Inorder_Traversal(Shin_Node: Shin_Node_Pt);
  begin
```

```
  if Shin_Node <> nil then
    begin
      Inorder_Traversal(Shin_Node^.Left_Son);
      if Shin_Node^.Key_Counter > 0 then
        begin
          write(Sort_Out_File, Shin_Node^.Node_Char, ':',Shin_Node^.Key_Counter:1,' ');
          Order_Count := Order_Count + 4;
          Temp_Cnt := Order_Count mod PRINT_LINE_DIVISOR;
          if (Temp_Cnt >= 0) and (Temp_Cnt <= 3) then writeln(Sort_Out_File);
        end
      else
        begin
          write(Sort_Out_File, Shin_Node^.Node_Char,' ');
          Order_Count := Order_Count + 2;
          if ((Order_Count mod PRINT_LINE_DIVISOR) = 0) or
              ((Order_Count mod PRINT_LINE_DIVISOR) = 1) then
            writeln(Sort_Out_File);
        end;
      Inorder_Traversal(Shin_Node^.Right_Son);
    end;
end;

procedure Postorder_Traversal(Shin_Node: Shin_Node_Pt);
begin
  if Shin_Node <> nil then
    begin
      Postorder_Traversal(Shin_Node^.Left_Son);
      Postorder_Traversal(Shin_Node^.Right_Son);
      if Shin_Node^.Key_Counter > 0 then
        begin
          write(Sort_Out_File, Shin_Node^.Node_Char, ':',Shin_Node^.Key_Counter:1,' ');
          Order_Count := Order_Count + 4;
          Temp_Cnt := Order_Count mod PRINT_LINE_DIVISOR;
          if (Temp_Cnt >= 0) and (Temp_Cnt <= 3) then writeln(Sort_Out_File);
        end
      else
```

```
            begin
              write(Sort_Out_File, Shin_Node^.Node_Char,' ');
              Order_Count := Order_Count + 2;
              if ((Order_Count mod PRINT_LINE_DIVISOR) = 0) or
                  ((Order_Count mod PRINT_LINE_DIVISOR) = 1) then
                writeln(Sort_Out_File);
            end;
        end;
end;

begin   {------------ Procedure Sort_Search_Keys starts from here ----------}
  No_Keys := 0;
  No_Out_Keys := 0;
  Keys_File_Name := 'SDB/Sort_Keys_File.txt';
  AssignFile(Sort_Keys_File, Keys_File_Name);
  reset(Sort_Keys_File);
  writeln(Data_Out, 'Sort starts from here.');
  AssignFile(Sort_Out_File, 'Data_Sort.txt');
  rewrite(Sort_Out_File);
  if Sort_Debug then
    begin
      writeln(Sort_Out_File);
      writeln(Sort_Out_File, '<---------- Shin sort starts from here ---------->');
      writeln(Sort_Out_File, ' Program reads key strings, stores them into Shin tree, and ');
      writeln(Sort_Out_File, ' performs preorder traversal to print every node value out.');
      writeln(Sort_Out_File);
    end;
  Key_File_Name := 'SDB/Search_Key_File.txt';
  AssignFile(Search_Key_File, Key_File_Name);
  reset(Search_Key_File);

  Root_Pt := nil;
  Header_LR_Flag := none;
  while not EOF(Sort_Keys_File) do
    begin
      No_Keys := No_Keys + 1;
```

```
    Read_and_Store_Key(Key_Pt);
    Shin_Sort_The_Key(Root_Pt, Key_Pt);
    if Sort_Debug then
      begin
        Order_Count := 0;
        Preorder_Traversal(Root_Pt);
        writeln(Sort_Out_File);
        writeln(Sort_Out_File,'------------------------------------------------------------');
      end;
  end;
if Sort_Debug then
  begin
    writeln(Sort_Out_File);
    Print_Input_List;
    writeln(Sort_Out_File);
    writeln(Sort_Out_File,'_____________________________________________________________');
    writeln(Sort_Out_File);
    writeln(Sort_Out_file,'Preorder traversal after ', No_Keys:1,' keys are inserted into the tree.');
    writeln(Sort_Out_File);
    Order_Count := 0;
    Preorder_Traversal(Root_Pt);
    writeln(Sort_Out_File);
    writeln(Sort_Out_File,'_____________________________________________________________');
    writeln(Sort_Out_File);
    writeln(Sort_Out_file,'Inorder traversal after ', No_Keys:1,' keys are inserted into the tree.');
    writeln(Sort_Out_File);
    Order_Count := 0;
    Inorder_Traversal(Root_Pt);
    writeln(Sort_Out_File);
    writeln(Sort_Out_File,'_____________________________________________________________');
    writeln(Sort_Out_File);
    writeln(Sort_Out_file,'Postorder traversal after ', No_Keys:1,' keys are inserted into the tree.');
    writeln(Sort_Out_File);
    Order_Count := 0;
    Postorder_Traversal(Root_Pt);
    writeln(Sort_Out_File);
```

```
      writeln(Sort_Out_File,'_____');

    end;
  if Sort_Debug then
    begin
      writeln(Sort_Out_File);
      writeln(Sort_Out_File);
      writeln(Sort_Out_File, '<---------- Printing Shin tree starts from here ---------->');
      writeln(Sort_Out_File, ' Traversing the tree in preorder, it will print out keys and');
      writeln(Sort_Out_File, ' show how stack is changed.  It will push a node character');
      writeln(Sort_Out_File, ' into the stack and pop one from the stack.  The following');
      writeln(Sort_Out_File, ' shows how a key character is inserted into and deleted from');
      writeln(Sort_Out_File, ' the stack.  A key will be printed out whenever collected');
      writeln(Sort_Out_File, ' stack items make a full key.');
      writeln(Sort_Out_File);
    end;
  Push_Pop_Counter := 0;
  Stack_Top := nil;
  if Sort_Debug then
    begin
      Print_Sorted_Keys(Root_Pt);
      Print_Input_List;
      Print_Output_List;
    end;
  if Sort_Debug then
    begin
      writeln(Sort_Out_File);
      writeln(Sort_Out_File);
      writeln(Sort_Out_File, '<---------- Shin search starts from here ---------->');
      writeln(Sort_Out_File, ' The following output will show what keys are looked for and');
      writeln(Sort_Out_File, ' how characters in a key are searched one after another in the');
      writeln(Sort_Out_File, ' tree.  The search result, found or not found, will be printed');
      writeln(Sort_Out_File, ' after all.');
      Search_Key_In_Tree(Root_Pt);
    end;
  CloseFile(Sort_Keys_File);
```

```
    CloseFile(Sort_Out_File);
  end;        {End of Procedure Sort_Search_Keys}
```