Ropes unit

Reference Guide

Version 1.0

Adriaan Rappard

Introduction

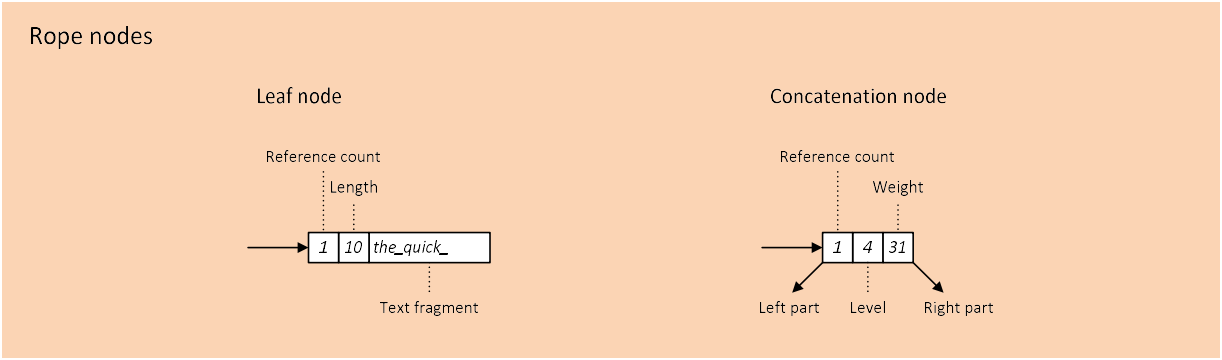
Most programming languages offer basic functionality for text processing where text is stored in memory as a consecutive sequence of characters. Two strategies are common to keep track of the number characters in the sequence. Either the sequence is terminated by a special symbol that may not occur within the sequence, or the number of characters is maintained explicitly by storing it in a separate location. In Pascal, as in many other programming languages, such a sequence is called a *character string*, or *string* for short. The number of characters in a string is called the *length* of the string.

Although storing a text as consecutive characters is fine for shorter texts this causes problems when texts become larger. One issue is how much memory to allocate for storing a text, the exact storage requirements are usually not known in advance. Allocating a large chunk of memory beforehand may be very wasteful in terms of required memory space, allocating a small chunk of memory requires moving the entire text to another location when the string outgrows the space available. This clearly is very wasteful in terms of processing effort. Similar issues occur with the insertion and deletion of text fragments in a large text since these also require moving large parts of the text from one memory location to another.

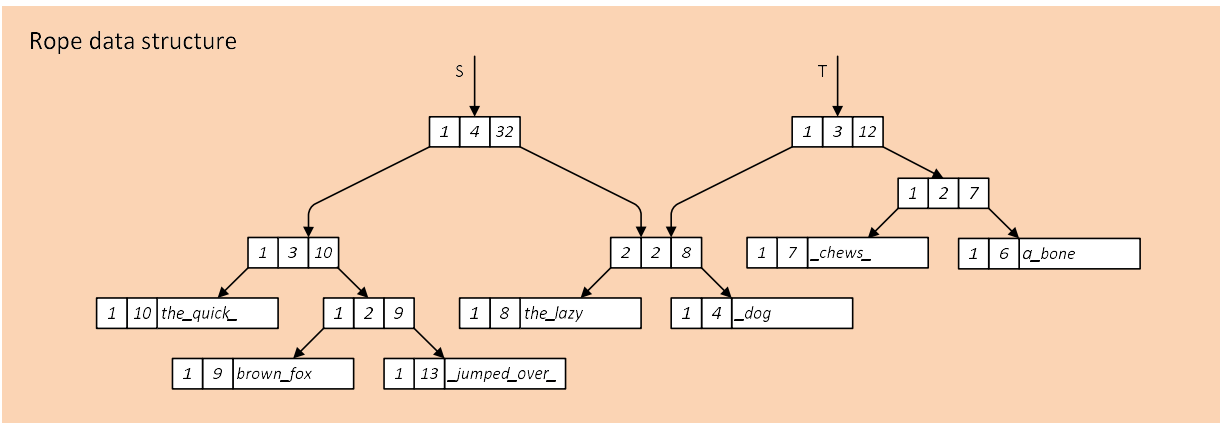
This document describes an implementation of the rope data structure, an alternative to strings that is better suited for processing large texts. The implementation is done as a unit for Free Pascal that defines a rope datatype that retains most of the familiar properties of strings. General information about the rope data structure can be found on https://en.wikipedia.org/wiki/Rope_(data_structure).

Rope data structure

The rope data structure stores text in a binary tree that consists of two types of nodes, leaf nodes and concatenation nodes. Each leaf node stores a text fragment as a consecutive sequence of characters and a length field containing the number of characters in the sequence. Each concatenation node contains references to two other nodes to form a larger text fragment, a level field, and a weight field. The level field holds the largest relative level of the node towards the leaves of both branches. The level of the top node is called the *depth* of the tree. The weight field holds the total number of characters stored in the left branch. Both types of node contain a reference count to keep track of the number of outstanding references to the node. The structure of both types of node is illustrated in the picture below.



Keeping track of the number of outstanding references to the rope data structure makes it possible to change a part of the text leaving the original unchanged. This behavior is often desired in word processing since it allows for recovering from erroneous changes in the text. The following picture gives an example. Here the text S: 'the quick brown fox jumped over the lazy dog' was changed into the text T: 'the lazy dog chews a bone' without destroying the original text.



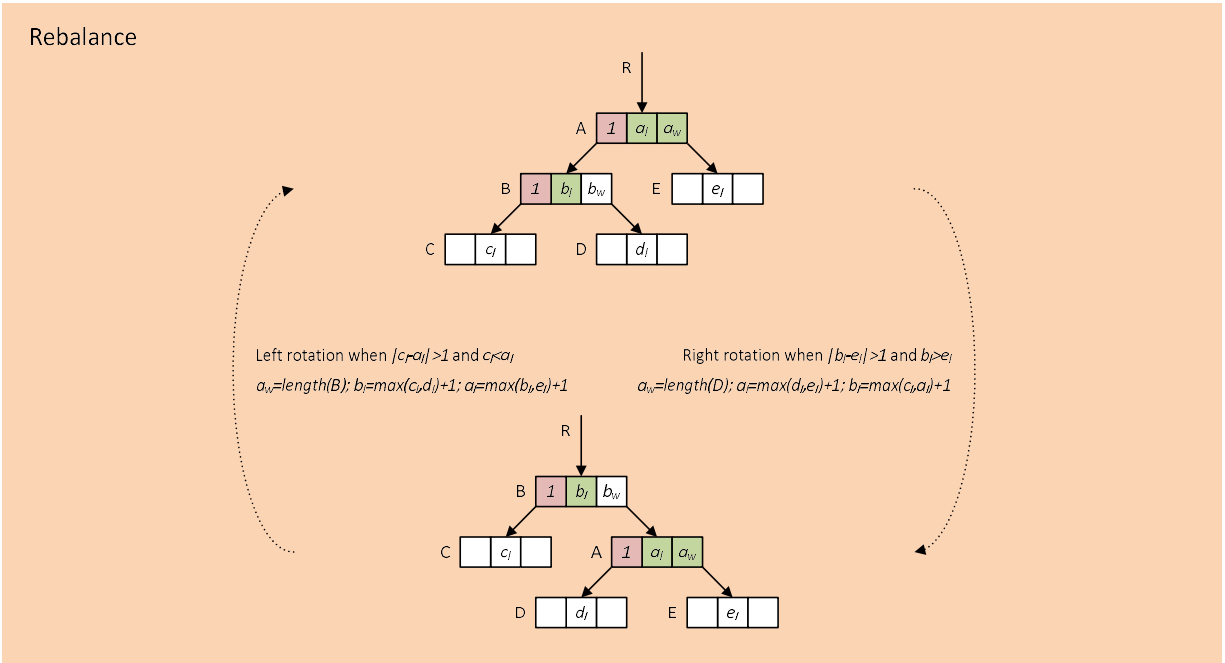
The rope data structure allows for text fragment insertion and deletion without requiring the movement of large parts of the text. Leaves that become fully filled can be transformed into a concatenation of two partially filled leaves. In reverse, a concatenation of two partially filled leaves can be transformed into a single leaf whenever the text is small enough to fit in a single leaf. When the leaves are relatively small this greatly reduces the need for moving large parts of the text.

The price paid for this is twofold. On the one hand there is an overhead in the total storage needed for storing the concatenation nodes. Leaves should not be too small to keep this overhead within reasonable bounds. Larger leaves will however increase the amount of data movement needed for splitting and merging leaves. On the other hand retrieving the address of a character at a certain position within the data structure requires more effort in comparison to strings since this requires traversing the tree from top to bottom in search of the leaf page where the desired character resides.

Balancing the tree

In order to keep the tree traversal that is needed to retrieve a character at a certain position efficient imbalance in the tree must be prevented. The tree is considered balanced when the depth of the left and right branch differs by no more than one in every node. Unfortunately, the tree loses balance when nodes are added or removed as text fragments are inserted or deleted. To restore the balance a *rebalance* operation is performed recursively at every node that is affected by the change. The *rebalance* operation compares the level of the two branches of the node and, if the level difference is greater than one, the tree is *rotated* in such a way that the node with the higher level becomes the new top node as illustrated in the picture below.

Note that reference count of the rotating nodes A and B, both marked red in the picture, must be one to ensure that these nodes are not referenced by more than one variable during rotation. The rebalance operation does not check this because rebalancing is only performed during rope concatenation on nodes which are guaranteed not to be referenced more than once. After rotating the nodes A and B the level and weight fields, marked green in the picture, are changed according to their new position in the tree.



More information about balanced binary trees can be found on https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree.

Primitive operations

The primitive operations on the rope data structure are *concatenation* and *split*. The concatenation operation is straightforward in principle and creates a new concatenation node that refers to both constituting parts. The actual implementation of concatenation however is more complex in order to keep the tree balanced. The split operation splits the rope into two ropes while keeping the original rope unimpaired. Both primitive operations and how they are performed are discussed in detail in the following paragraphs.

More familiar operations like *insert* and *delete* are constructed by making use of the primitive operations. The *insert* operation for example splits the rope using the *split* operation after which it concatenates the left part, the text to insert, and the right part together by using the *concatenation* operation twice. Likewise the *delete* operation uses the *split* operation twice to split the rope in tree parts. Then it concatenates the first and last part using the *concatenation* operation and discards the middle part.

Concatenation

The concatenation operation concatenates two ropes together into one large rope. In order to ensure that the resulting rope is balanced the concatenation operation distinguishes between three cases depending on the depth difference of both ropes. The way concatenation is performed in each case is illustrated in the following picture. In the picture the new nodes that are created during the concatenation are marked green, the changes to existing nodes are marked red.

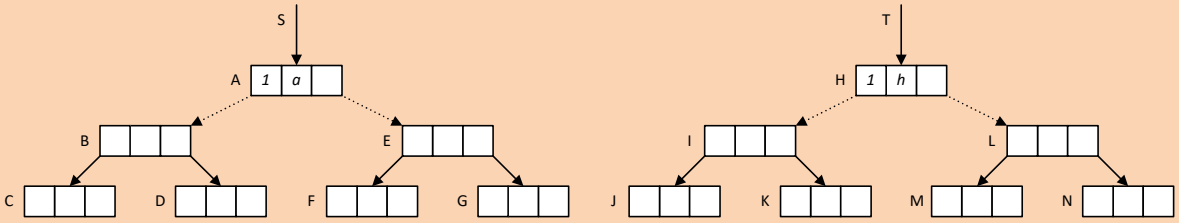
When both ropes are of equal depth ($a_l = h_l$) then concatenation is straightforward. In this case concatenation is performed by creating a new top node and adjusting the reference counts in the top nodes marked A and H of the ropes being concatenated.

When the depth of the left rope is smaller than the depth of right rope ($a_l < h_l$) then the left branch of the right rope is followed until the node marked J with equal depth ($a_l = j_l$) is found. During this process new concatenation nodes are created to form a new tree that references the right parts of the nodes encountered. The reference count of the nodes that are now also referenced by the new tree are incremented. Note that only the first newly created node marked R, one intermediate node marked U, and the last two nodes node marked Y and Z are drawn. Finally the left branch of the node created last marked Z is directed to top node of the left rope marked A. During the creation process of the new tree the appropriate values for the level and weight fields of the new concatenation nodes are computed as indicated by the formulae depicted next to the nodes.

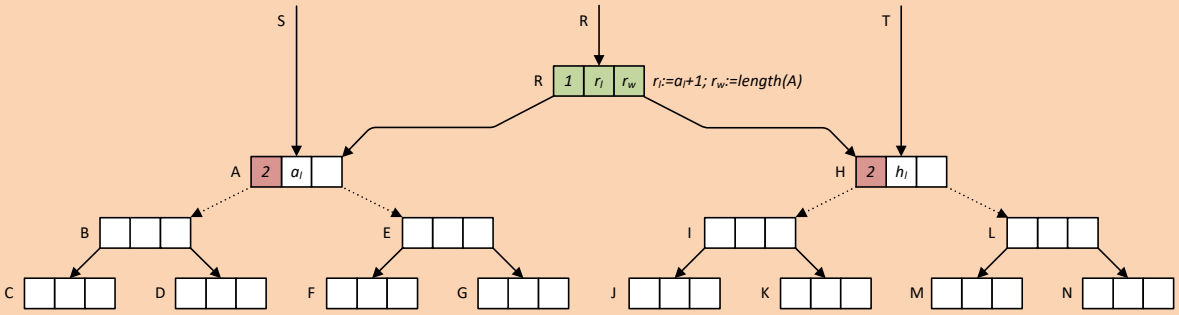
When the depth of the left rope is greater than the depth of right rope ($a_l > h_l$) then the concatenation process is the mirror image of the one described directly above. Now the right branch of the left rope is followed until the node marked G with equal depth ($a_l = g_l$) is found. New concatenation nodes are created to form a new tree that references the left parts of the nodes encountered. Finally the right branch of the node created last marked Z is directed to top node of the right rope marked H.

Concatenation

Before concatenation

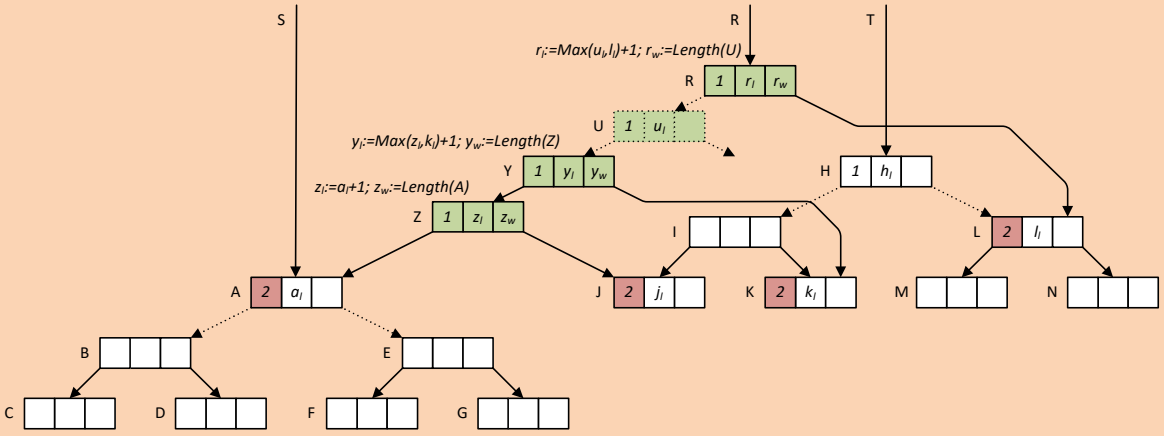


When $a_l=h_l$: Insertion of a new node



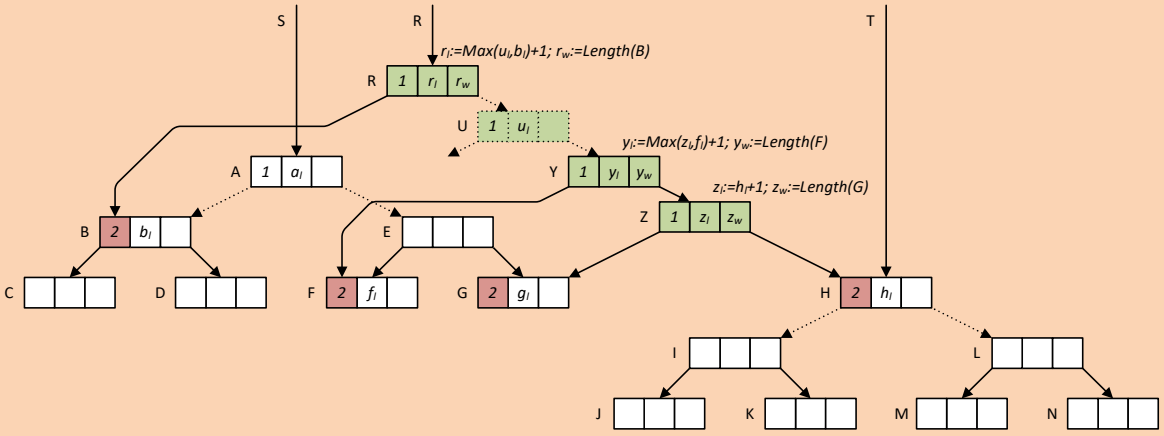
When $a_l<h_l$: recursive concatenation in left branch of H

Create new rope following left branch of H until $a_l=j_l$



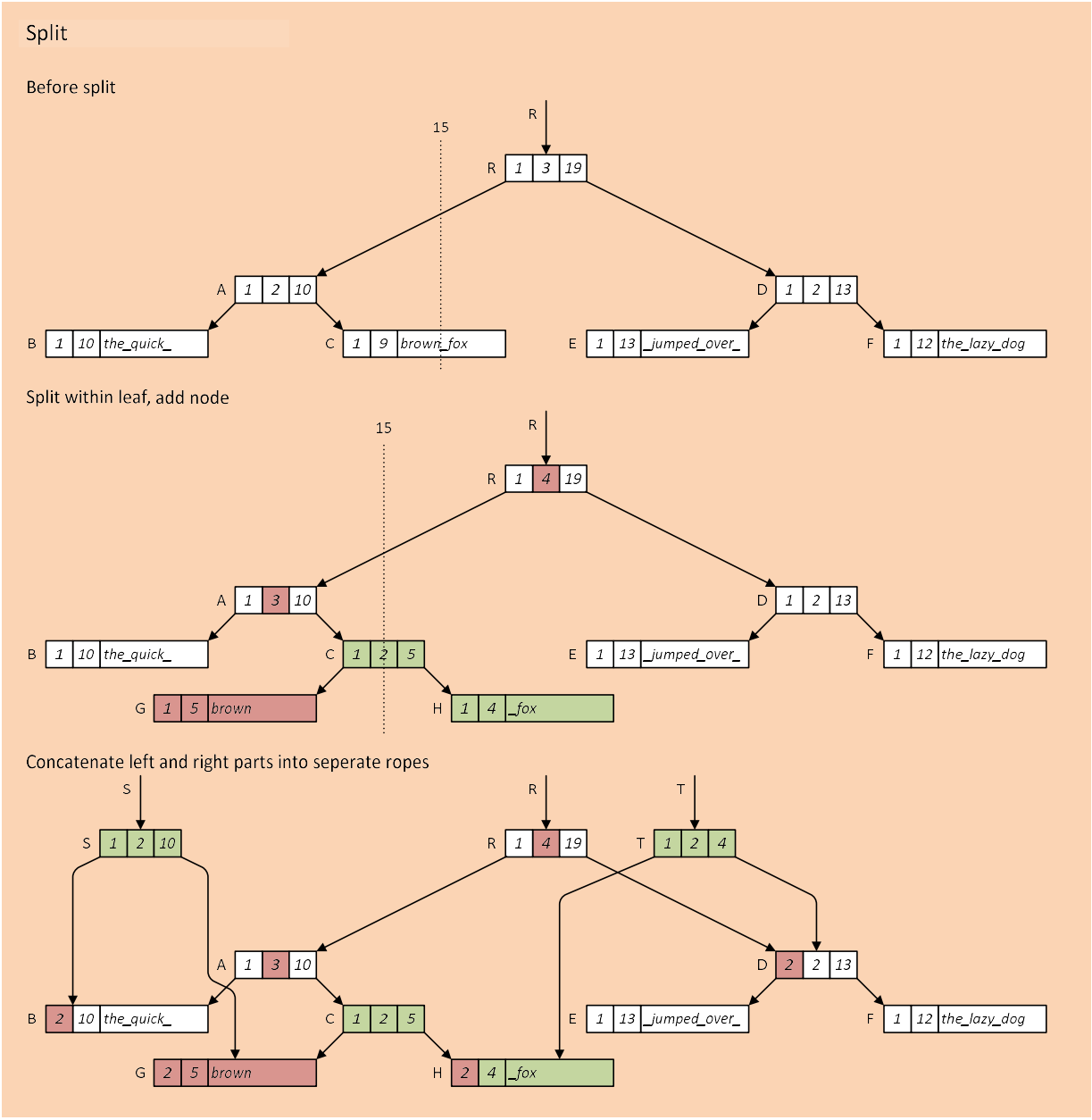
When $a_l>h_l$: recursive concatenation in right branch of A

Create new rope following right branch of A until $h_l=g_l$



Split

The split operation splits a rope into two parts while keeping the original rope unimpaired as illustrated in the picture below. The split operation traverses the tree recursively from the top node marked R downwards each time selecting the left or right branch of the concatenation nodes encountered. When the split index is smaller than the weight of the concatenation node then the left branch is followed. When the split index is greater or equal than the weight then the weight is subtracted from the split index after which the right branch is followed.



This process stops when a leaf node is encountered. When the split index lies within the leaf the leaf node is split by creating a new leaf node, copying the text beyond the split index to the new leaf node, and truncating the content of the leaf node that was found by adjusting the length field. The truncated leaf node is then concatenated with the new leaf node leaving the original rope text unimpaired. In the example depicted the left branch was followed at the top node marked R to arrive

at the node marked A. Next the right branch of node A is followed to arrive at the leaf marked C. This leaf node is then split into two leaves marked G and H, and the two parts are concatenated together by concatenation node C replacing the original leaf.

Finally the path taken while traversing the tree is followed in the reverse order. During this process new trees for the left and right parts of the original rope are constructed by concatenating the nodes referenced by the left or right branches of the nodes into new ropes marked S and T. In nodes where the right branch was followed the node referenced by the left branch is concatenated into the left rope S, in the nodes where the left branch was followed the node referenced by the right branch is concatenated into the right part tree marked T. During the creation process of the new trees the appropriate values for the level and weight fields of the new concatenation nodes are set.

Using the Ropes unit

The `Ropes` unit implements the rope data structure as a Pascal data type. In order to make the unit easy to use the operations are designed to resemble the familiar `String` operations as closely as possible. Since outstanding references to every rope must be counted ropes are somewhat more difficult to use than ordinary strings. The way these reference counts are maintained resembles the bookkeeping that is performed while using `AnsiString` variables. The main difference is that it is now the responsibility of the programmer to call the appropriate bookkeeping routines. With `AnsiString` variables, these calls are automatically inserted by the compiler.

To understand how this bookkeeping is done consider the following program.

```
// Bookkeeping routines example

program RopeExample1;

   uses Ropes;

   var R:Rope;

   procedure Reverse(var R:Rope);

      var S:Rope; I:Cardinal; Ch:Char;

   begin {Reverse(var R:Rope)}
   Make(S);
   for I:=1 to Length(Load(R)) do
      begin
      Ch:=Select(Load(R),I);
      Store(S,Load(Ch)+Load(S))
      end;
   Store(R,Load(S));
   Drop(S)
   end; {Reverse(var R:Rope)}

begin {RopeExample1}
Make(R);
Store(R,Load('Hello world!'));
WriteLnRope(Load(R));
Reverse(R);
WriteLnRope(Load(R));
Drop(R)
end. {RopeExample1}
```

In order to properly initialize rope variables the program must explicitly create the variables by calling the procedure `Make` for each rope variable declared. Normally this call is issued just before

the first statement of the block in which the variable is declared. Also, since storage for rope nodes is allocated on the heap, rope variables must be explicitly destroyed by calling the procedure `Drop` before the declared variable goes out of scope, i.e. just after the last statement of the block. Note that variables of a structured data type containing ropes must issue a call to `Make` and `Drop` for every rope element in the structure. Also note that dynamic variables containing ropes must issue a call to `Make` for every rope element they contain after the dynamic variable is created. Similarly a call to `Drop` must be issued for every rope element before disposing the dynamic variable.

When rope variables are used in expressions the rope's value is retrieved by issuing a call to the `Load` function. The value retrieved creates a new implicit reference to the rope and the `Load` function takes care of incrementing the reference count to account for the implicit reference. Note that the Load function must not be called when passing a variable as a parameter by reference i.e. as an argument for a variable parameter.

Assignment of rope variables is handled by the `Store` procedure that takes care of destroying the old rope value thus freeing the storage allocated for it. The Store procedure also takes care of decrementing the reference count of new rope value that is passed as an argument.

Finally, the `Load` function is overloaded so it can accept an `AnsiString` as its argument. This overloaded function converts the string argument to a rope tree.

The following program shows how bookkeeping is done when ropes are used as value parameters and/or a function result.

```
// Bookkeeping routines example

program RopeExample2;

   uses Ropes;

   var R:Rope;

   function Reverse(R:Rope):Rope;

      var S:Rope; I:Cardinal; Ch:Char;

   begin {Reverse(R:Rope):Rope}
   Declare(R); Make(Reverse); Make(S);
   for I:=1 to Length(Load(R)) do
      begin
      Ch:=Select(Load(R),I);
      Store(S,Load(Ch)+Load(S))
      end;
   Store(Reverse,Load(S));
   Drop(S); Drop(R); Pass(Reverse)
   end; {Reverse(R:Rope):Rope}

begin {RopeExample2}
Make(R);
Store(R,Load('Hello world!'));
WriteLnRope(Load(R));
Store(R,Reverse(Load(R)));
WriteLnRope(Load(R));
Drop(R)
end. {RopeExample2}
```

Value parameters behave like local variables that are already initialized so they do not have to be initialized by a call to the `Make` procedure. The `Declare` procedure is called instead in order to keep track of the total number of active rope variables and the total length of all ropes.

The function result must be initialized by a call to the `Make` procedure just like a variable. Since the value must be preserved it can't be dropped. A call to the `Pass` procedure must be issued instead in order to keep track of the total number of active rope variables and the total length of all ropes.

Constants and types

Constants

The `Ropes` unit defines the following constants.

Maximum number of characters stored in a leaf node.

```
MaxLLen = 256
```

Maximum depth of a rope tree.

```
MaxLevel = 64
```

Empty rope constant.

```
Empty : Rope = (Nil)
```

Types

The `Ropes` unit defines the following types.

Allowed node level values, an empty rope has level 0, a leaf node has level 1, concatenation nodes can have a level ranging from 2 to `MaxLevel`.

```
Level = 0..MaxLevel
```

Pointer to a generic node i.e. either a leaf or a concatenation node.

```
PGeneric = ^GenericNode
```

Pointer to a leaf node.

```
PLeaf = ^LeafNode
```

Pointer to a concatenation node.

```
PConcat = ^ConcatNode
```

The Rope data type is declared as a one element array with a single pointer to a generic node. This somewhat complicated structure is necessary to allow overloading the operators that would normally act on the pointer value.

```
Rope = array [0..0] of PGeneric
```

Generic node type used to determine if a node is a leaf or a concatenation node. The node type is encoded in the most significant bit of the `TRCount` field. This bit contains a 0 for leaf nodes and a 1 for concatenation nodes. The other bits of the `TRCount` field hold the reference count of the node.

```
GenericNode = record
   // Type tag and reference count
   TRCount     : Cardinal
   end; {GenericNode}
```

Leaf node type used to store character content. The `TRCount` field contains a 0 in the most significant bit to indicate a leaf node. The other bits of the `TRCount` field hold the reference count

of the node. The `LLen` field holds the number of characters stored in the `LText` field starting at the first position of the array.

```
LeafNode = record
   // Type tag and reference count
   TRCount     : Cardinal;
   // Leaf length: number of characters in leaf text
   LLen        : 0..MaxLLen;
   // Leaf text character buffer
   LText       : array [1..MaxLLen] of Char
   end; {LeafNode}
```

Concatenation node type used to build the rope tree. The `TRCount` field contains a 1 in the most significant bit to indicate a concatenation node. The other bits of the `TRCount` field hold the reference count of the node. The `NLevel` field holds the largest relative level of the node towards the leaves of both branches. The `NWeight` field holds the total number of characters stored in the left branch.

```
ConcatNode = record
   // Type tag and reference count
   TRCount     : Cardinal;
   // Node level within rope tree
   NLevel      : Level;
   // Node weight: length of text in left branch
   NWeight     : Cardinal;
   // Left and right branches
   Left, Right : Rope
   end; {ConcatNode}
```

Enumeration class type to allow the implementation of an enumerator operator. The enumerator operator is not used in explicit expressions but enables the use for..in loops with a rope variable. The fields, properties and methods are used automatically from within for..in loop.

```
REnumerator = class
   private
      // Back reference to rope
      BRef        : Rope;
      // Current position and rope length
      Pos, Len    : Cardinal;
      // Get character at current position
      function GetCurrent:Char;
   public
      // Construct enumerator
      constructor Create(R:Rope);
      // Destruct enumerator
      destructor Destroy; override;
      // Character at current position
      property Current:Char read GetCurrent;
      // Advance to next position enumerator
      function MoveNext:Boolean;
      end; {Renumerator}
```

Procedures, functions , and operators

The Ropes unit defines the following procedures, functions, and operators. These come into two categories, *functional core* routines and *support and diagnostic* routines.

Functional core routines

The Make procedure initializes rope variables and increments the rope variable count.

```
// Create empty rope variable
procedure Make(var R:Rope);
```

The Drop procedure decrements the rope variable count and adjusts the overall rope length. Then it decrements the reference count of the node referenced by R. When the reference count reaches zero it disposes all unreferenced nodes of the rope.

```
// Drop rope variable
procedure Drop(var R:Rope);
```

The Declare procedure increments the rope variable count and adjusts the overall rope length.

```
// Declare rope value parameter
procedure Declare(var R:Rope);
```

The Pass procedure decrements the rope variable count and adjusts the overall rope length.

```
// Pass function result
procedure Pass(var R:Rope);
```

The Load function with a Rope argument increments the reference count of the node referenced by R and returns a reference to this node.

```
// Allocate rope variable and retrieve value
function Load(var R:Rope):Rope;
```

The Load function with a String argument constructs a new rope from the string text.

```
// Create rope value from ANSI string
function Load(Str:AnsiString):Rope;
```

The Store procedure decrements the reference count of the node referenced by R. When the reference count reaches zero it disposes all unreferenced nodes of this rope. Finally the reference in S is copied into R and the overall rope length is adjusted.

```
// Release rope variable and store new value
procedure Store(var R:Rope; S:Rope);
```

The Select function returns the character that is stored at position I from rope R.

```
// Select and return value of R[I]
function Select(R:Rope; I:Cardinal):Char;
```

The + operator with Rope operands returns the concatenation of the two operands.

```
// Concatenate ropes S and T
operator + (S, T:Rope) R:Rope;
```

The = operator with Rope operands returns True when the content of both operands is equal.

```
// Return true when rope S equal T
operator = (S, T:Rope) R:Boolean;
```

The <> operator with `Rope` operands returns `True` when the content of both operands is not equal.

```
// Return true when rope S unequal T
operator <> (S, T:Rope) R:Boolean;
```

The < operator with Rope operands returns `True` when the content of the first operand is strictly less than the content of the second operand in alphabetic order.

```
// Return true when rope S strictly less than T
operator < (S, T:Rope) R:Boolean;
```

The > operator with `Rope` operands returns `True` when the content of the first operand is strictly greater than the content of the second operand in alphabetic order.

```
// Return true when rope S strictly greater than T
operator > (S, T:Rope) R:Boolean;
```

The <= operator with `Rope` operands returns `True` when the content of the first operand is less than or equal to the content of the second operand in alphabetic order.

```
// Return true when rope S less than or equal T
operator <= (S, T:Rope) R:Boolean;
```

The >= operator with `Rope` operands returns `True` when the content of the first operand is greater than or equal to the content of the second operand in alphabetic order.

```
// Return true when rope S greater than or equal T
operator >= (S, T:Rope) R:Boolean;
```

The `Length` function with a `Rope` argument returns the number of characters in the rope content.

```
// Return length of rope R
function Length(R:Rope):Cardinal;
```

The `Length` function with a `String` argument returns the number of characters in the string content. This function is redefined to allow normal use of the overloaded function.

```
// Return length of string S
function Length(S:AnsiString):Cardinal;
```

The `Pos` function with `Rope` arguments returns the position of the first occurrence of the content of rope `S` in the content of rope `R`. The function returns zero when the content of `S` does not occur in the content of `R`.

```
// Return position of first occurrence of rope S in rope T
function Pos(S, T:Rope):Cardinal;
```

The `Concat` function with two `Rope` arguments returns the concatenation of the two operands.

```
// Concatenate ropes R1 and R2
function Concat(R1, R2:Rope):Rope;
```

The `Concat` function with three `Rope` arguments returns the concatenation of the three operands.

```
// Concatenate ropes R1, R2 and R3
function Concat(R1, R2, R3:Rope):Rope;
```

The Concat function with four Rope arguments returns the concatenation of the four operands.

```
// Concatenate ropes R1, R2, R3 and R4
function Concat(R1, R2, R3, R4:Rope):Rope;
```

The Copy function with a Rope argument returns a rope of length L of which the content is a copy of the content of rope R, starting at position I. When L is larger than the length of rope R, the result is truncated. When the position I is larger than the length of rope R, then an empty rope is returned.

```
// Return rope of L characters from R starting at R[I]
function Copy(R:Rope; I, L:Cardinal):Rope;
```

The Delete procedure with a Rope argument removes L characters from rope R, starting at position I, and the length of the rope is adjusted. Note that the characters after the removed characters are now located at a position that lies L places to the left of their original position.

```
// Delete L characters from R starting at R[I]
procedure Delete(var R:Rope; I, L:Cardinal);
```

The Insert procedure with a Rope argument inserts the content of rope S in rope R, starting at position I, and the length of the rope is adjusted. Note that the characters after the inserted characters are now located at a position that lies L places to the right of their original position.

```
// Insert rope S in rope R after R[I]
procedure Insert(S:Rope; var R:Rope; I:Cardinal);
```

The Upcase function with a Rope argument returns a rope that contains a copy of the content of rope R in which all lower case letters are converted to upper case.

```
// Return conversion of R to upper case
function Upcase(R:Rope):Rope;
```

The Lowercase function with a Rope argument returns a rope that contains a copy of the content of rope R in which all upper case letters are converted to lower case.

```
// Return conversion of R to lower case
function Lowercase(R:Rope):Rope;
```

The ReadRope procedure creates a rope R from the characters it reads from standard input. The characters are read from standard input until the end of the line is encountered. The line termination is considered a terminator and is not part of the rope content.

```
// Read rope text from standard input
procedure ReadRope(var R:Rope);
```

The ReadLnRope procedure with a Rope variable parameter is almost identical to the ReadRope procedure. The only difference is that the ReadRope procedure stops at the end of the line whereas the ReadLnRope procedure proceeds to the first character of the next line after reading.

```
// Read rope text and line terminator from standard input
procedure ReadLnRope(var R:Rope);
```

The `ReadRope` procedure with an extra `Text` file variable parameter is almost identical to the `ReadRope` procedure. The only difference is that now input not read from standard input but from the `Text` file instead.

```
// Read rope text from text file
procedure ReadRope(var T:Text; var R:Rope);
```

The `ReadLnRope` procedure with an extra `Text` file variable parameter is almost identical to the `ReadLnRope` procedure. The only difference is that now input not read from standard input but from the `Text` file instead.

```
// Read rope text and line terminator from text file
procedure ReadLnRope(var T:Text; var R:Rope);
```

The `WriteRope` procedure writes the content of rope `R` to standard output.

```
// Write rope text to standard output
procedure WriteRope(R:Rope);
```

The `WriteLnRope` procedure writes the content of rope `R` to standard output followed by a line terminator.

```
// Write rope text and line terminator to standard output
procedure WriteLnRope(R:Rope);
```

The `WriteRope` procedure with an extra `Text` file variable parameter is almost identical to the `WriteRope` procedure. The only difference is that now output is not written to standard output but to the `Text` file instead.

```
// Write rope text to text file
procedure WriteRope(var T:Text; R:Rope);
```

The `WriteLnRope` procedure with an extra `Text` file variable parameter is almost identical to the `WriteLnRope` procedure. The only difference is that now output is not written to standard output but to the `Text` file instead.

```
// Write rope text and line terminator to text file
procedure WriteLnRope(var T:Text; R:Rope);
```

Support and diagnostic routines

The `enumerator` operator is not used in explicit expressions but enables the use for..in loops with a rope variable.

```
// Enumerator operator to allow for .. in loops
operator enumerator (R:Rope):REnumerator;
```

The `LeafNodes` function with a `Rope` argument returns the number of leaf nodes of the rope. Note that some or all of these nodes may be shared with other ropes.

```
// Return rope leaf node count for rope R
function LeafNodes(R:Rope):Cardinal;
```

The `ConcatNodes` function with a `Rope` argument returns the number of concatenation nodes of the rope. Note that some or all of these nodes may be shared with other ropes.

```
// Return rope concatenation node count for rope R
function ConcatNodes(R:Rope):Cardinal;
```

The `Depth` function returns the level of the top node of the rope i.e. the largest relative level of this node towards the leaves of both branches.

```
// Return rope tree depth for rope R
function Depth(R:Rope):Level;
```

The `RopeVariables` function returns the number of rope variables created within the current scope of execution of the program.

```
// Return current rope variable count
function RopeVariables:Cardinal;
```

The `LeafNodes` function returns the total number of leaf nodes created within the current scope of execution of the program.

```
// Return total leaf node count for current rope variables
function LeafNodes:Cardinal;
```

The `ConcatNodes` function returns the total number of concatenation nodes created within the current scope of execution of the program.

```
// Return concatenation node count for current rope variables
function ConcatNodes:Cardinal;
```

The `Length` function returns the sum of the lengths of all rope variables created within the current scope of execution of the program. Note that this sum may exceed the actual number of bytes of storage that is actually used since different ropes may share part of their structure.

```
// Return total length of current rope variables
function Length:Cardinal;
```

Rope/String comparison

Theoretically ropes have an advantage over strings when processing large texts in terms of speed when inserting or deleting parts of the text. But there is a price to be paid in terms of storage overhead and the time needed to reach an individual character at a certain position. In order to facilitate making the choice between using ropes or strings two tests were performed comparing a rope and a string implementation.

The tests compare the time needed using two types of ropes, one with the standard leaf size of 256 bytes and one with a larger leaf size of 512 bytes, with the time needed when using strings. In order to be able to use very long texts the `AnsiString` data type was used as the string data type. The memory utilization of both rope types was determined also for comparison. The memory utilization of the string implementation was not determined since the documentation does not provide enough insight in the exact mechanics of the `AnsiString` data type.

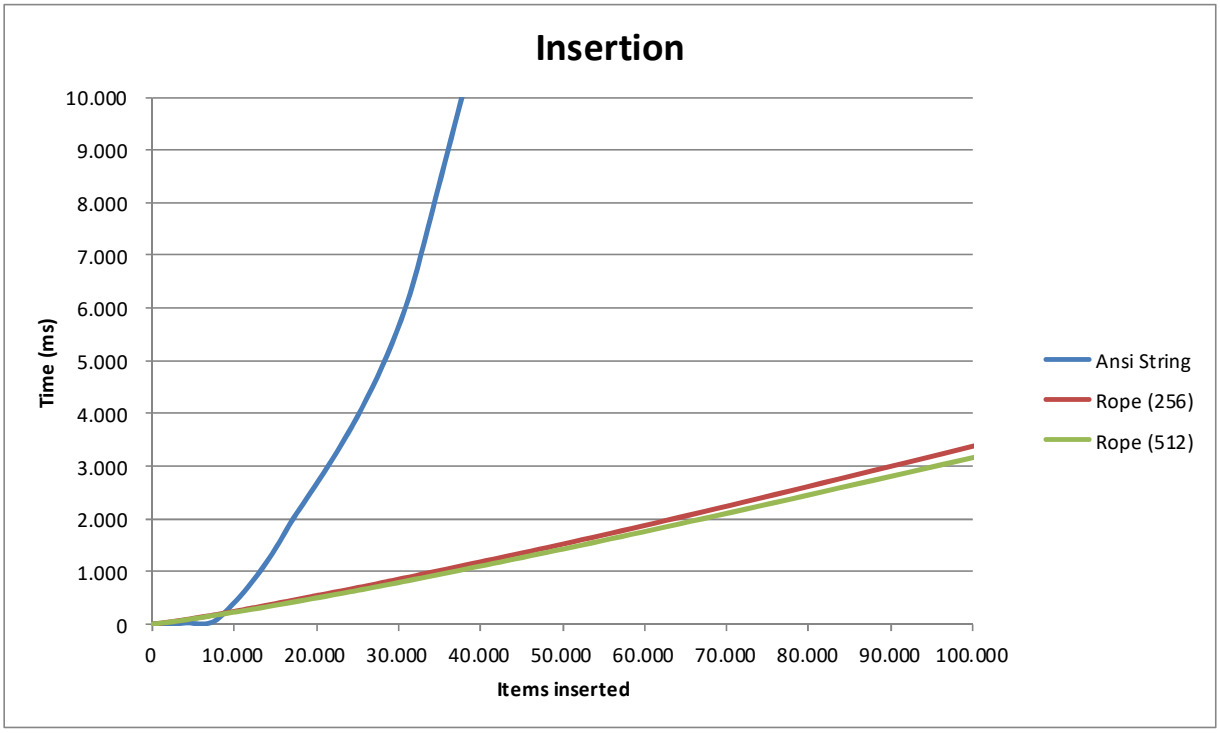
Both tests use a long text that is built by repeatedly concatenating a string that holds a single sentence. Note that, since the text is built from a string rather than a rope, this gets around the reference counting mechanism so the resulting text now stores multiple distinct copies of the sentence.

Scan and insertion

The scan and insertion test uses the sentence 'The quick fox jumped over the lazy dog.' as the base sentence for constructing the text. After the text is built a search is performed for all occurrences of the word 'fox' and at every position found the word 'brown' with a trailing space is inserted at that position. The following table shows the results of repeating the test for texts of increasing length.

	Ansi String	Rope				
		Leaf size 256 bytes			Leaf size 512 bytes	
Repetition	ΔT (ms)	ΔT (ms)	Utilization (%)		ΔT (ms)	Utilization (%)
1	0	0	18		0	9
2	0	0	35		0	18
4	0	0	69		0	35
8	0	0	66		0	69
16	0	0	87		0	68
32	0	0	73		0	90
64	0	0	78		1	89
128	0	1	78		1	89
256	1	3	80		3	89
512	2	7	79		7	91
1.024	2	16	79		17	91
2.048	5	38	79		35	91
4.096	24	86	79		80	92
8.192	127	191	79		181	92
16.384	1.800	429	79		395	92
32.768	7.024	942	79		874	92
65.536	32.468	2.073	79		1.946	92
131.072	130.440	4.560	79		4.263	92

The following graphic shows the relation between the time needed and the number of insertions.

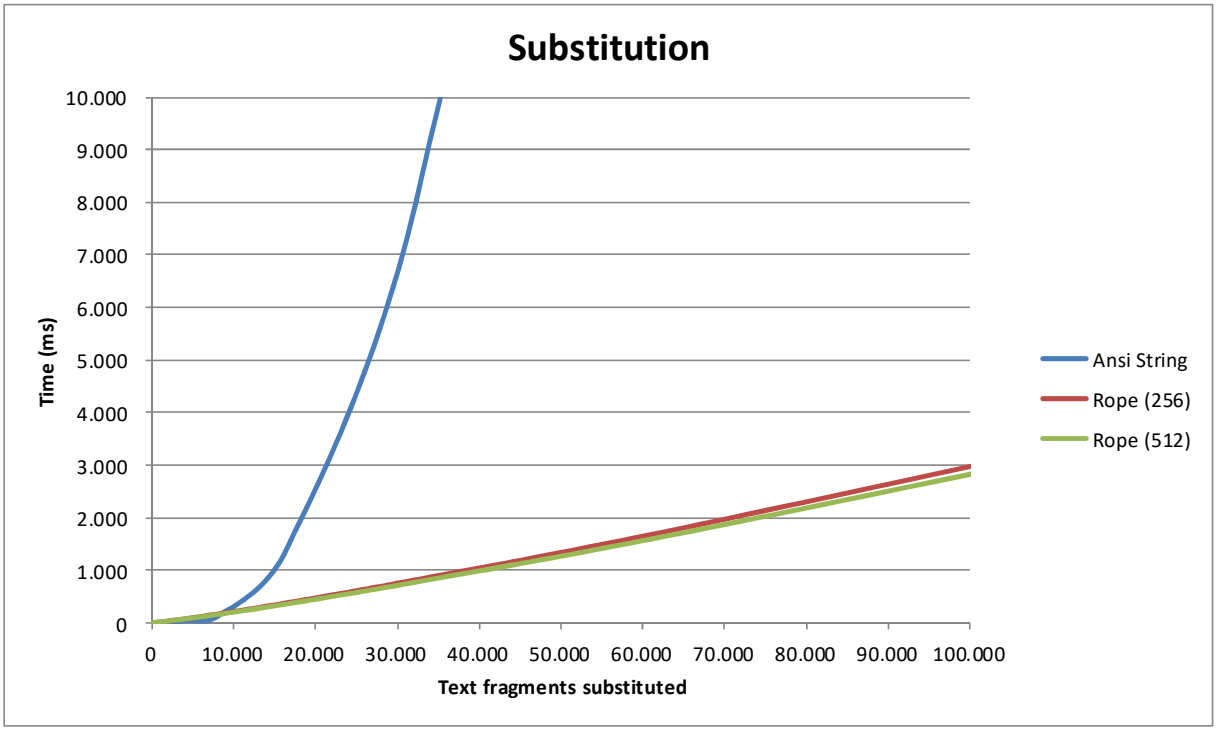


Scan and substitution

The scan and substitution test uses the sentence 'The quick brown fox jumped over the lazy dog.' as the base sentence for constructing the text. After the text is built a search is performed for all occurrences of the word 'brown' and at every position found the word 'brown' is substituted by the word 'red'. The following table shows the results of repeating the test for texts of increasing length.

	Ansi String	Rope				
		Leaf size 256 bytes		Leaf size 512 bytes		
Repetition	ΔT (ms)	ΔT (ms)	Utilization (%)	ΔT (ms)	Utilization (%)	
1	0	0	18	0	9	
2	0	0	34	0	17	
4	0	0	66	0	34	
8	0	0	63	0	66	
16	0	0	83	0	65	
32	0	0	82	0	86	
64	0	1	81	1	85	
128	0	2	81	1	85	
256	0	3	83	1	85	
512	1	7	83	6	87	
1.024	2	15	83	14	88	
2.048	5	34	83	33	87	
4.096	42	76	83	73	87	
8.192	141	170	83	161	88	
16.384	1.383	376	83	355	88	
32.768	8.349	831	83	786	88	
65.536	35.963	1.823	83	1.729	88	
131.072	147.736	4.020	83	3.819	88	

The following graphic shows the relation between the time needed and the number of substitutions.



The RopeDemo program

In order to demonstrate how the `Ropes` unit can be used for word processing the `RopeDemo` program shows an implementation of a small text editor. Central in the design is an array that holds the text being edited along with control information like cursor position, current line and column, etcetera. When a change is requested the editor first creates a copy of the contents of the current array slot in the next slot. Then the editor advances to the next slot which contains the copy and applies the change. This design greatly facilitates implementation of an 'undo' and a 'redo' operation. Copying the entire text however would not be very efficient using strings since the entire text is duplicated with every change that is applied. When using ropes the overhead of copying is much less due to the use of reference counts within the data structure. With ropes only the nodes that are affected by the change are duplicated.

The `RopeDemo` program divides the screen into three areas, a header area that shows the name of the file, a content area containing the text , and a footer area that displays editor status information. The status display can be switched between cursor, text, history, storage, and node information.

- The cursor information shows the line, column, and relative position of the cursor within the text.
- The text information shows the number of lines, characters, and pages of the text.
- The history information shows the number of undo entries, redo entries, and total entries in the editor history.
- The storage information shows the total length of all ropes and the bytes used for storage. The memory utilization is calculated from these and the result is shown as a percentage. Since the ropes share nodes the memory utilization may exceed 100%.
- The node information shows the number of concatenation nodes, leaf nodes, and total nodes within the system.

The `RopeDemo` program recognizes the normal keys used for scrolling and editing text (*arrow keys*, *PgUp*, *PgDn*, *Tab*, *Delete*, *Backspace*). The *Home* and *End* key move the cursor to the beginning and end of the current line respectively. The *Insert* key toggles between inserting and overwriting characters. The commands recognized by the `RopeDemo` program are listed in the following table.

Command	Function
Alt N	Create new file
Alt O	Open existing text file
Alt S	Save file
Alt W	Save file under another name
Alt B	Mark beginning of selection
Alt E	Mark end of selection
Alt C	Copy selected text fragment to clipboard
Alt P	Paste text fragment from clipboard
Alt F	Find text fragment
Alt R	Find and replace text fragment
Alt A	Search next occurrence
Alt U	Undo change
Alt D	Redo change
Alt X	Terminate and exit program
Alt 1	Switch to cursor information
Alt 2	Switch to text information

Command	Function
Alt 3	Switch to history information
Alt 4	Switch to storage information
Alt 5	Switch to node information

The `RopeDemo` program was set up to use a display area of 80 columns by 50 lines so the actual display device used should accommodate these dimensions to run the program without alterations. In a windowed environment the window must be set up to correspond to these dimensions in order to display information correctly. In Microsoft Windows this is done by right-clicking the top edge of the window and setting the properties accordingly.