Design and implementation of a compressed linked list library

Yoran Heling

Design and implementation of a compressed linked list library

Final project

Company supervisor:
Anton Prins
D&R Elektronica

Author:
Yoran Heling

University supervisor:
Gerard Nanninga
Hanze University Groningen

June 2, 2010

Preface

This report is written as part of my final project during the Electrical Engineering course at the Hanze University Groningen, and represents the research and findings from the project *Implementation and design of a compressed linked list library.* This project is commissioned by D&R Electronics.

This report is targeted at people with an interest in compression techniques and digital data processing, and for software engineers working with applications that handle large amounts of data in application memory. Basic knowledge in the fields of computer architecture and software design is required in order to fully understand the topics being discussed in this report.

I would like to thank ing. A. Prins of D&R Electronics and ir. G.J. Nanninga of the Hanze University of Groningen for their constructive feedback and continued supervision during this project.

Gieten, June 2010.

Contents

Abstract

This project aims to decrease the working memory of applications working with large amounts of data by designing and implementing a library that allows in-memory compression of data that is ordered in linked lists. This project is tested and optimized for the open source disk usage analyzer "ncdu", which is written in C for UNIX-like operating systems.

Data compression is the technique of reducing the storage requirements of (digital) data by removing redundant information. Many different compression algorithms exist, each with their own strengths and weaknesses. Simple but weak algorithms are Delta coding and Run-length encoding. Stronger algorithms, such as Lempel-Ziv, Huffman coding and Range encoding, provide better compression but at the cost of slower compression or decompression. However, as many free and well-tested general-purpose compression libraries have already been written, it is more efficient to use one of those than implement compression by ourselves.

A linked list is a popular method of creating data structures and is used in a wide variety of applications. Linked lists consist of a number of nodes, each of which has one or more links or references to other nodes. Each node contains a fixed number of elements in which the information is stored. By linking all nodes together in a certain layout or with a certain system, it is possible to create complex data structures. Common structures are single-dimension lists and hierarchical trees.

Because nodes in a linked list are usually quite small and compression works best on larger amounts of data, a method has to be found to store nodes in larger blocks of memory. Two such methods have been researched: grouping whole nodes together in blocks of memory or grouping the elements of the nodes together. While grouping elements together generally provides better compression than grouping nodes, the difference is only very small. In terms of performance, both methods are expected to be fast enough for our purpose. The method of grouping nodes together was chosen because it was less complex and had significantly less overhead from keeping track of meta data.

Performing compression or decompression is a relatively expensive task, and should be avoided as much as possible. For this reason, a few memory blocks remain in memory in an uncompressed state. Which blocks remain in memory is decided by a block replacement algorithm. After comparing several existing algorithms, it was decided that the Least Recently Used (LRE) strategy was the most suitable for use in the library.

The functionality of the library is exposed to the application through a minimal API. This API has an initialization function, with which some configuration parameters can be specified. The functions for creating and deleting nodes are similar to the C functions `malloc()` and `free()`. Reading and writing node data can be done using two separate functions. The header file of the library also includes useful macros to help developers with debugging their applications.

The implemented library uses an efficient approach to store the nodes within memory blocks so that only little overhead is required even for larger block sizes, while still providing

all the information necessary to keep track of used and unused nodes and to merge and split nodes. Memory blocks are internally managed in two arrays: one array holding information about every block in memory, and a separate array for the uncompressed blocks. The complex functionality of the library is internally divided into a number of functions that each perform a more simple task.

The library has been implemented in ncdu and was tested for performance and compression with practical usage of the application. These primarily test results indicate that the use of the library slows down most operations with a factor 4 to 6. The memory usage with use of the library is around 8 times lower than without, which is an overall compression ratio of about 20%.

Although there is still room for further optimisations and improvements, the designed and implemented library meets the initially specified requirements, and can be used in practical applications.

1 Introduction

This chapter is an introduction to the project, describes our problem and includes a solution analysis.

1.1 Introduction

This report represents the research and findings of the final project of the course Electrical Engineering at the Hanze University Groningen, and is commissioned by D&R Electronics.

This project aims to decrease the working memory of applications working with large amounts of in-memory data by using compression on linked lists data types. More specifically, this project will be tested and optimized for the disk usage analyzer "ncdu".

1.2 Problem specification

"ncdu" is an open source application written for UNIX-like operating systems, which can make an analyzation on the disk usage patterns of a file system. In order to do this, it needs store a relatively large amount of data in memory. Due to its nature as a hard drive analysis tool, using the hard drive itself as temporary storage is not an option – this will render it impossible to analyze a file system with too little free space left. Being a general purpose tool, used on both embedded systems with limited available RAM and on large servers running memory-hogging database systems, ncdu has no choice but to make as efficient use of the available memory as possible.

As many other applications do, ncdu internally uses "linked lists" to represent its data structure in memory. A linked list typically consists of a large amount of small "nodes", where each node has one or more "links" to other nodes, thus forming a data structure. Linked lists are very popular in software applications due to their ease of use and the endlessly complex data structures they can represent.

As a means of decreasing the memory requirements of ncdu – and possibly of other applications as well – we can design a library that implements a compressed linked list. This library will provide the application with the functions necessary to create and work with any type of linked list, while the nodes of the list are managed internally by the library and are stored compressed in memory. This library has the following requirements:

A. The library shouldn't be limited to only one type of linked list; it should be possible to create any data structure.

B. The nodes in the linked list can be of a variable size. Although the size of each node should be known at the time it is created.

C. For linked lists with many nodes, the memory used to hold the list should be considerably less when created with the library than when created using traditional methods.

D. The linked list must be stored entirely in application memory, temporary storage on a hard disk is not allowed.

E. The library should have support for all basic operations: reading, modifying, creating and deleting nodes.

F. While the performance for most operations on the compressed linked list will undoubtedly be slower than on a traditional (uncompressed) linked list, it should still be within acceptable levels. "Acceptable" in this context means that there should be no noticeable performance degradation with normal usage of ncdu.

G. The library should be professionally packaged and documented.

As it is impossible to create a library that offers both the best possible compression and the highest possible performance, these two factors will need to be carefully weighed against each other in the design, in order for the library to provide reasonably good compression with an acceptable performance.

1.3 Plan of action

This project can be divided in three stages: analysis, design and implementation.

The first stage, analysis, will focus on specifying and describing our problem. The technologies at hand will also be discussed in this stage, explaining common compression techniques and what linked lists are and how they can be used.

The second stage, design, will answer the questions related to the several design issues concerning the creation of a compressed linked list. For a large part, the focus will be on how to store the nodes of a linked list into memory. There are two major ways to do this: grouping several *nodes* together in blocks of memory, or grouping the *elements* of each node together. Both methods will be considered and their advantages and disadvantages will be weighed against each other in order to find the best method for the library, especially with regards to compression ratio and performance. Because only a part of all the nodes can be stored uncompressed in memory at a time, an algorithm for determining which nodes to compress and decompress at which point in time ("demand paging") will need to be decided upon. This stage will also look at which compression algorithm to use, and an API will be designed so that an application can make use of the library.

In the last stage the library will be implemented and tested, using ncdu as a testing platform.

Table 1.1 gives a first rough estimate of how long each stage will take and when each subject will be worked on. The numbers are the week numbers since the start of the project, week 1 in the schedule is week 6 in the year calendar. An estimation of the time distribution of the three stages is given in figure 1.1. A log of the actual project progress is kept in appendix B. The chapters in this report are in chronological order and are written while working on each subject.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Analysis																				
Problem description																				
Introduction to compression																				
Introduction to linked lists																				
Design																				
Storage: grouping nodes																				
Storage: grouping elements																				
Demand paging																				
API																				
Implementation																				
Library																				
Updating and testing ncdu																				

Table 1.1. Rough estimate of the project schedule per week

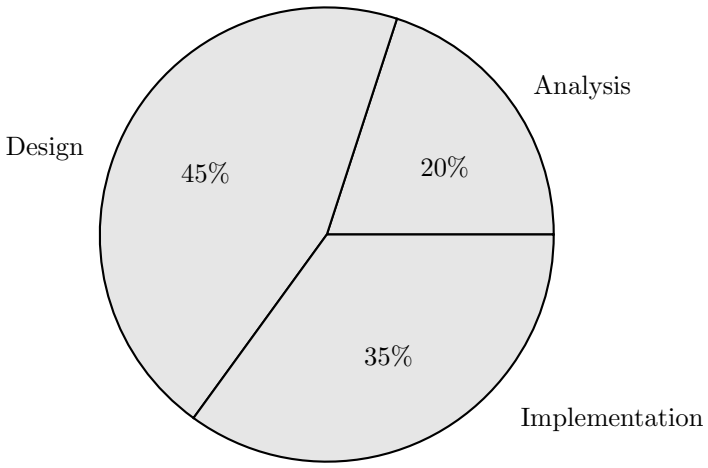

Figure 1.1. Estimated time distribution among the three stages

2 Introduction to compression

Data compression is the technique of reducing the storage requirements of (digital) data by removing redundant information. Compression is widely used in file formats and network protocols, and there exist many different compression algorithms, each having their strengths and weaknesses.

Many popular file formats use compression. Images are often stored in the PNG or JPEG formats. Both provide strong compression on image data, resulting in small file sizes. The MP3 and OGG formats are specialized in compressing audio data and are thus very popular for encoding and distributing music.

Compression algorithms can be grouped in two major categories: lossless and lossy. With lossless compression, it does not matter how many times you compress or decompress the data, the data constructed after decompressing (output) will always be exactly the same the data before compressing (input). This is not true for lossy compression: the output may not be fully equivalent to the input. Lossy compression can result in the distortion of data or the loss of precision, but usually provides better compression ratios than lossless algorithms.

As loss of data is generally not acceptable for intermediate data storage, where the data is likely to be decompressed, modified and later recompressed again more than once, the library will be limited to using a lossless compression algorithm. In order to get a feel on how compression works and what kinds of data may compress better than others, this chapter will clarify some famous lossless compression algorithms.

2.1 Delta coding

Delta coding[Bou98] is a simple technique in which, instead of absolute values, the differences to an earlier value is stored. If you have a large list of numbers where each number only differs slightly from the previous number, instead of storing each number in full, you can also calculate the difference with the previous number and store that instead.

Consider the following example: you have a list of 6 numbers (table 2.1) where the absolute values of these numbers occupy 5 characters. The storage of this list without compression will be $6 * 5 = 30$ characters. With delta encoding, you only need to store the absolute value of the first number, for all subsequent numbers you only have to store the difference with the previous number. Because the differences between the numbers is relatively small, the delta encoded list requires significantly less space. The storage required to hold the same list after compression is now only $5 + (5 * 1) = 10$ characters, this is a factor of 3 times smaller than the uncompressed list.

Due to its simplism, delta coding is trivial to implement in most applications. It also provides relatively good compression ratios for data that does not change much, real life

#	0	1	2	3	4	5
original data	50000	50002	50003	50010	50010	50019
delta coded	50000	2	1	7	0	9

Table 2.1. Example of compression using delta coding

examples of this are (audio) signals and other data sampled at fixed time intervals. For all other kinds of data, however, delta coding may not really help much. In fact, if the differences between each value are very large, delta coding may result in larger storage requirements than simply storing the data without applying compression.

2.2 Run-length encoding

Another technique to decrease the storage requirements of data is Run-length encoding (RLE) [Bou98]. With run-length encoding, long sequences of the same character are encoded as a single character and a number indicating how many times that character is repeated.

For example, the string "AAAABCCC" starts with four times an 'A', followed by one 'B' and ends with three times a 'C'. With RLE, this can be encoded as "4A1B3C" (table 2.2), reducing the total number of characters from 9 to 5. Notice how the single 'B' does not have a number asociated to it. This means it should not be repeated, but just copied over once.

#	0	1	2	3	4	5	7	8
original data	A	A	A	A	B	C	C	C
encoded	4A				B	3C		

Table 2.2. Example of compression using run-length encoding

As with delta coding, RLE is very easy to implement and compression and decompression can be done very fast. It's usefulness, however, is only limited to data in which many characters are repeated without other intermediate characters. RLE is mostly used for image data, where for simple images many subsequent pixels have the same color values and are thus suitable for compression using this method. For other types of data it is not commonly used.

2.3 Lempel-Ziv

A more efficient and more commonly used compression technique is the Lempel-Ziv[McF92, l.7] algorithm, named after its authors Abraham Lempel and Jacob Ziv. This algorithm encodes (ranges of) symbols as references to earlier used data. This way, repetitions and patterns in the uncompressed data can be detected and encoded in smaller representations.

A Lempel-Ziv encoded string consists of two types of symbols: raw, uncompressed data that is directly copied from the original data, and references to earlier used symbols. References consist of a negative offset referring to the position of the start of the data, and a length that specifies how long the sequence is.

Consider the 10 characters long string "ABCCCABBCC". This can be encoded using Lempel-Ziv as shown in table 2.3. A character enclosed in single quotes represents an

uncompressed character and references are represented as two numbers within brackets, where the first number is the offset relative to the current position and the second number the length. Because the first three characters are all new, these are literally copied into the encoded data. Character #3 and #4, however, are repetitions of character #2, and are thus replaced with a single reference with offset -1 (so starting at character position $3-1=2$) and has a length of 2. Notice how this reference overlaps itself: Character #3 refers to #2, while #4 refers to #3. Because the decoder copies all data sequentially, this is not a problem. Characters #5 and #6 are a repetition of #0 and #1, and #7 to #9 are a repetition of #1 to #3, so these can also be replaced with references. If you encode all unmodified characters and references with a fixed size, you'll notice that the storage requirements for this example string has been reduced from 10 symbols to just 6.

#	0	1	2	3	4	5	6	7	8	9
original data	A	B	C	C	C	A	B	B	C	C
encoded	'A'	'B'	'C'	(-1,2)		(-5,2)		(-6,3)		

Table 2.3. Example of compression using Lempel-Ziv

One of the main advantages of Lempel-Ziv compression is that decompression can be done incredibly fast: a decoder can restore the original data simply by sequentially copying over all unmodified characters and expanding the references by copying data from the (partially) decompressed data. Decompression is thus only slightly slower than performing a normal copy of the (uncompressed) data.

As seen in the example, Lempel-Ziv works especially well on data with a lot of repetition, and is therefore relatively popular as a general purpose compression algorithm, used in archive file formats like ZIP and gzip.

A disadvantage of Lempel-Ziv is that the compression algorithm is quite complex and not really fast. A significant amount of comparison operations and pattern matching is required in order to find the most optimal encoded sequence. Most implementations limit themselves in how far they look back into the data to find patterns. This significantly improves the compression speed, but at the cost of a less optimal compression ratio.

2.4 Huffman coding

With Huffman coding[Ast04], all (usually fixed-size) character codes are replaced with variable-length codes, where the length depends on how often the character appears in the data. Because applications need to be able to differentiate between individual characters, variable length encoding has the disadvantage that there should be some way to indicate how long each character is, often requiring an extra data field or special character separation codes. Huffman encoding, however, solves this problem by encoding the characters in a tree-like structure.

Take for example the tree in figure 2.1, here we have a simple encoding table for the characters A, B and C. The C is represented in one bit ('0'), while A and B are represented using two bits ('11' and '10', respectively). With this tree we can encode the previous example string "ABCCCABBCC" as shown in table 2.4. The encoded data has now been reduced to 15 bits, this is more than a factor of 5 times smaller than the original data, assuming each character was originally encoded using 8 bits.

Of course, if you need to encode more than 3 characters, the tree will need to be expanded

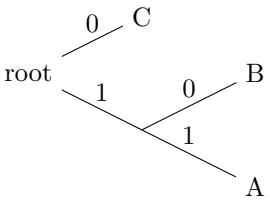

Figure 2.1. Example tree to encode the characters 'A', 'B' and 'C'

#	0	1	2	3	4	5	6	7	8	9
original data	A	B	C	C	C	A	B	B	C	C
encoded	11	10	0	0	0	11	10	10	0	0

Table 2.4. Example of compression using Huffman coding

and the length of the bit sequences will also increase. However, you still have the freedom to choose how long the encoding for each individual character will be, and by making sure that the characters with a higher probability of occurring in the original data are coded in shorter bit sequences than characters with a lower probability, you can still achieve pretty good compression.

In order to be able to decode Huffman-coded data, an application has to know which tree was used to encode the data. For some applications a static tree is used: all data is simply encoded with the same tree and the decoder only works with that specific tree. This way there is no need to store the tree separately alongside the compressed data. In practice, however, this is rarely done, as it is extremely important that the tree used for encoding is optimized for the data, in order to get the best compression ratio.

A downside to traditional Huffman coding is that, in order to construct an optimal tree for a specific chunk of data, the entire chunk of data will need to be analyzed first. This makes the compression stage slower because a full pass on the data needs to be done twice – once to create the tree and a second time to do the actual encoding – and in some cases this may not even be possible because the data is simply not known yet (for example, when you want to compress something while it is being downloaded from the internet).

To counter this, Adaptive Huffman code[LH87, Ch.4] was invented. Using this method, the tree is constructed while the data is being compressed, the encoded data will contain additional instructions for the decoder to modify or expand the tree while decoding. In addition to faster and sequential encoding, this has the added side effect that the tree can be modified to adapt to changes in the data, resulting in even better compression ratios than when one fixed tree is used for a larger chunk of data.

A strong advantage of Huffman code is that both compression and decompression can be done relatively fast. Compression and decompression consist simply of replacing characters using a lookup table. A lookup table with statistics will have to be maintained while compressing, making compression a bit slower than decompression, but this can still be done quite efficiently.

2.5 Range encoding

Range encoding is a more complex and more mathematical compression technique which encodes several characters into a single number. It works similar to Huffman coding in the sense that the data to be encoded has to be analyzed first in order to create a lookup table, which in turn has to be known to both the encoder and the decoder. Unlike Huffman coding, the characters are not ordered into a tree but are instead kept in an ordered lookup table where the frequency of the character in the data determines the size of the range which is 'allocated' for that character.

This is best explained with an example: consider the 5 characters long string "HELLO". The characters H, E and O all appear only once, and thus have a frequency of $1/5 = 20\%$. The L has a frequency of 40% because it appears twice. With this it is possible to create a lookup table as seen in table 2.5.

Character	Times used	Frequency
H	1/5	20%
E	1/5	20%
L	2/5	40%
O	1/5	20%

Table 2.5. Lookup table for "HELLO"

Next, an initial range of numbers is chosen to start off with. To avoid having to work with real numbers, the range is chosen such that it is likely large enough to hold the data, for this example the range 0 – 1.000.000 is used. This range is then divided into several sub-ranges, each assigned to a character. The size of each sub-range depends on the frequency of the character. See step 1 in table 2.6. The first character of the input data was 'H', which is the first sub-range (0 – 20.0000). This range is used in step 2, and the process is repeated until the last range is found, which is 71.360 – 72.000.

Step 1 (range 0 – 1.000.000)

H	20%	0 - 200.000
E	20%	200.000 – 400.000
L	40%	400.000 – 800.000
O	20%	800.000 – 1.000.000

Step 2 (range 0 – 200.000)

HH	20%	0 – 40.000
HE	20%	40.000 - 80.000
HL	40%	80.000 – 160.000
HO	20%	160.000 – 200.000

Step 3 (range 40.000 – 80.000)

HEH	20%	40.000 – 48.000
HEE	20%	48.000 – 56.000
HEL	40%	56.000 - 72.000
HEO	20%	72.000 – 80.000

Step 4 (range 56.000 – 72.000)

HELH	20%	56.000 – 59.200
HELE	20%	59.200 – 62.400
HELL	40%	62.400 - 68.800
HELO	20%	68.800 – 72.000

Step 5 (range 68.800 – 72.000)

HELLH	20%	68.800 – 69.440
HELLE	20%	69.440 – 70.080
HELLL	40%	70.080 – 71.360
HELLO	20%	71.360 - 72.000

Table 2.6. Finding the range to encode "HELLO"

This range can be represented in binary in 17 bits as 10001011011000000 – 10001100101000000. However, because only the *range* is important and not the absolute numbers, the last 7 bits – which are the same for both numbers – do not have any significance and can be removed, after which the 10 bits number 1000101101 is left. This number uniquely represents our original data "HELLO", and can be decoded by using a similar algorithm, provided the decoder uses the same lookup table.

Range encoding is generally more effective than Huffman coding because the probability or frequency of a character does not have to be a power of two, as there is no fixed number of bits used for each character. It does, however, come at the cost of a more complex implementation and slower compression and decompression times.

2.6 Comparison

A comparison of the previously discussed compression algorithms is listed in table 2.7, outlining for which kind of data the algorithm is best suited in addition to the advantages and disadvantages of each technique.

Algorithm	**Advantages**	**Disadvantages**
Delta coding	Simple, efficient, good compression for data that does not change much.	Only suited for a specific type of data, little to no compression for other data.
RLE	Simple and fast, well suited for data with many repeated characters.	Only suited for a specific type of data.
Lempel-Ziv	Uses pattern matching, which provides good compression for data that has repetitions. Fast decompression.	(Relatively) slow and complex compression.
Huffman coding	Uses probability analysis, which provides good compression for data where a few characters occur more often than others. Relatively fast compression and decompression.	Requires the building and distributing of a special lookup table.
Range encoding	Uses probability analysis, suitable for the same kind of data as Huffman coding, but provides better compression.	More complex implementation and slower compression and decompression times.

Table 2.7. Comparison of compression algorithms

2.7 General-purpose compression libraries

In practice it rarely pays off to implement a compression algorithm on your own, because many free and well-tested compression libraries exist. These libraries have been thoroughly reviewed, tested and optimized by other software developers, making it unlikely that implementing your own will result in faster or better compression.

A few popular open source compression libraries are listed in table 2.8. With compression there is always a compromise between compression ratio and performance, the primary target column indicates for which the library has been optimized.

Library	Implemented algorithms	Primary target	Website
zlib	Lempel-Ziv, Huffman	Speed	`http://zlib.net/`
bzip2	RLE, Huffman, delta coding, block sorting[1]	Good compression	`http://bzip.org/`
LZO	Lempel-Ziv	Speed, low memory usage	`http://www.oberhumer.com/opensource/lzo/`
LZMA	Lempel-Ziv, range encoding	Good compression, fast decompression	`http://7-zip.org/sdk.html`

Table 2.8. Comparison of free general-purpose compression libraries

A comparison of the memory usage for (de)compression is shown in table 2.9. The minimum and maximum configurable memory requirements are compared. The libraries generally provide better compression ratios when configured to use more memory. It is interesting to note that LZO does not require any additional temporary storage for decompression. With LZMA the memory usage is highly dependant on the used dictionary size, which can be configured in the range from 4kB to 1GB.

	Compression		Decompression	
Library	**min.**	**max.**	**min.**	**max.**
zlib	2	640	1	32
bzip2	1200	7600	500	3700
LZO	8	64	-	-
LZMA	6190	> 10000	20	> 10000

Table 2.9. Memory requirements for compression libraries (in kilobytes)

[1] Block sorting hasn't been discussed in this chapter and isn't really a compression method: it re-orders the data in such a way that the common compression algorithms can provide much better compression.

3 Introduction to linked lists

A linked list is a popular method of creating data structures and is used in a wide variety of applications. Linked lists consist of a number of *nodes*, each of which has one or more *links* or *references* to other nodes. By linking all nodes together in a certain layout or with a certain system, it is possible to create complex data structures.

Each node contains a fixed number of *elements* in which the information is stored. In terms of the C programming language, a node is in fact nothing more than a structure in which at least one of the elements is a pointer to an instantiation of the same structure (also called a *recursive structure*). Listing 3.1 is an example of a linked list definition in C.

```
// Each variable of this type is a "node"
struct list_node {
  // These are elements of the node
  int number;
  char alpha;
  // This is a reference to the next node in the list
  struct list_node *next;
};
```

Listing 3.1. Example linked list definition

3.1 Common types of linked lists

The simplest type of a linked list is a "singly linked list". This is a linked list where each node has only one reference to another node: the next node in the list. The example in listing 3.1 is a singly linked list.

Figure 3.1 displays a singly linked list with three such nodes. In a C program, linked lists are referred to by a pointer to the first node, this pointer is called the *head*. The list can be traversed by taking the head node and following the *next* reference until this reference is *NULL*, indicating the end of the list. Unlike with arrays, this is the only method of finding a node in the list; random access is not possible within linked lists.

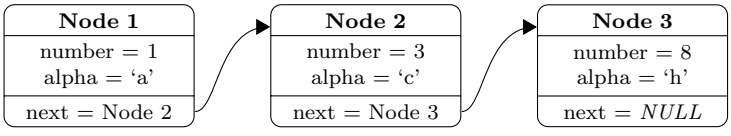

Figure 3.1. Example of a singly linked list

However, a strong advantage of linked lists when compared to regular arrays is that linked lists are far more flexible; nodes can be inserted or removed with a small number of operations. Suppose we want to insert a node between Node 1 and Node 2: this can be done by allocating memory for the new node, setting the *next* element to point to Node 2, and the *next* element of Node 1 to point to the newly created node.

Removing nodes from the list can be done even faster: you only have to set the value of the *next* element of the previous node to that of the next node, and then deallocate the memory of the node you want to remove. Changing the order in which the nodes appear in the list is also a simple matter of updating the *next* elements and the *head* pointer.

Singly linked lists work well for applications that often perform modifications to the list and only need to traverse the list sequentially. In some situations, however, you also want to traverse the list in the reverse order. While it is possible to do this with singly linked lists by constantly starting at the head to find the node which has *next* pointing to the current node (thus being the previous node), this is highly inefficient. A better alternative is to use "doubly linked lists": these have, in addition to the *next* element, also a *previous* element (often shortened to *prev*), referring to the previous node in the list. Figure 3.2 displays how the previous example can be implemented as a doubly linked list.

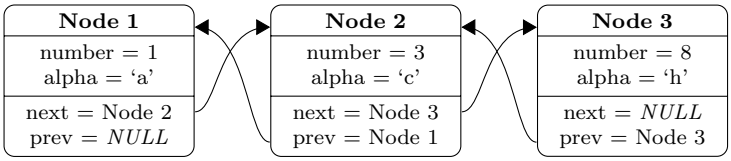

Figure 3.2. Example of a doubly linked list

While doubly linked lists make traversing the list easier, most operations that modify the list are slightly more expensive because two references have to be updated for each node, rather than only one. A doubly linked list also requires more memory in order to store the *prev* element.

Another common type of linked list is a "circular linked list", these are linked lists where the last node points back to the first node, thus creating a circular list. Both singly linked lists and doubly linked lists can be circular, with singly linked list the *next* element of the last node points back to the first node, and with doubly linked lists the *prev* element of the first node links to the last node. Figure 3.3 displays a circular singly linked list.

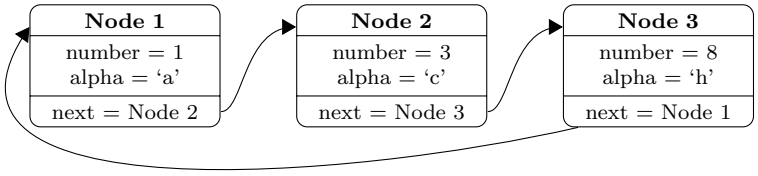

Figure 3.3. Example of a circular singly linked list

With circular linked lists, the *head* pointer can point to any of the nodes, as there is strictly speaking no "begin" or "end". This can simplify the implementation of some algorithms.

3.2 Tree data structures

Linked lists aren't only limited to single-dimension lists; they can also be used for more complex data structures.

Binary trees are a good example of this. Consider the Huffman tree discussed in chapter 2.4, this can be implemented using a linked list-like structure as seen in figure 3.4. Each node has two links to its *child nodes*, which can in turn have two child nodes as well, thus forming a tree-like data structure. The entire tree can be accessed from the *root*, which is Node 1 in our example.

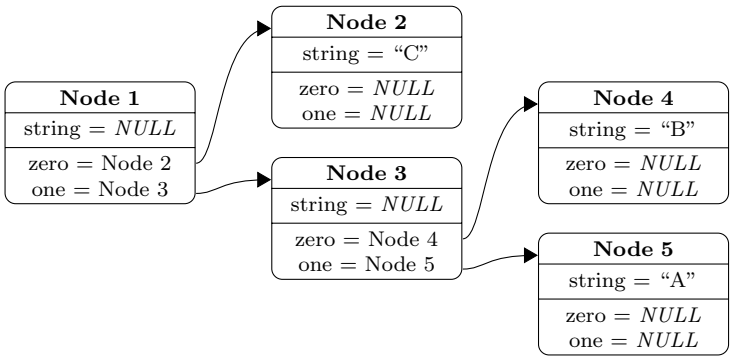

Figure 3.4. Example of a binary tree

Files and directories have a hierarchical structure similar to binary trees, and can as such be implemented using linked lists with relative ease. The main difference between a directory structure and a binary tree is that directories can have more than two child nodes. This can be realized by representing the contents of a single directory using a basic linked list: each node in this list then represents a file or subdirectory. Instead of the *zero* and *one* references in binary trees, each node will then have a single reference to the head of this linked list.

To make certain operations easier, it is possible to add a reference from each node back to its parent node. This reference is similar to the *previous* reference in a doubly linked list, as it allows an application to traverse the directory structure in reverse. This is useful, for example, to efficiently determine the full path to any file from only the reference to the node itself.

Most filesystems have support for allowing the same physical file to have multiple names or to be present in more than one directory. This is known as a "hard link". To implement these hard links in the same data structure, it is possible to link them together with a circular linked list. The advantage of using a circular linked list over a linear linked list is that the circular list does not require a *head* pointer, making it possible to find all linked files when all you have is a reference to only one of the nodes.

Figure 3.5 gives an example of a directory structure implemented using such a linked list. This structure represents the directory '/home/user', which contains a 'README' and 'ChangeLog' file and two directories: 'src' and 'doc'. 'README' and 'doc/README' are the same file, as indicated by the cirular linked list formed by linking the *hlnk* references together.

This is the data structure that ncdu uses to hold information and statistics of all the files and directories it has found.

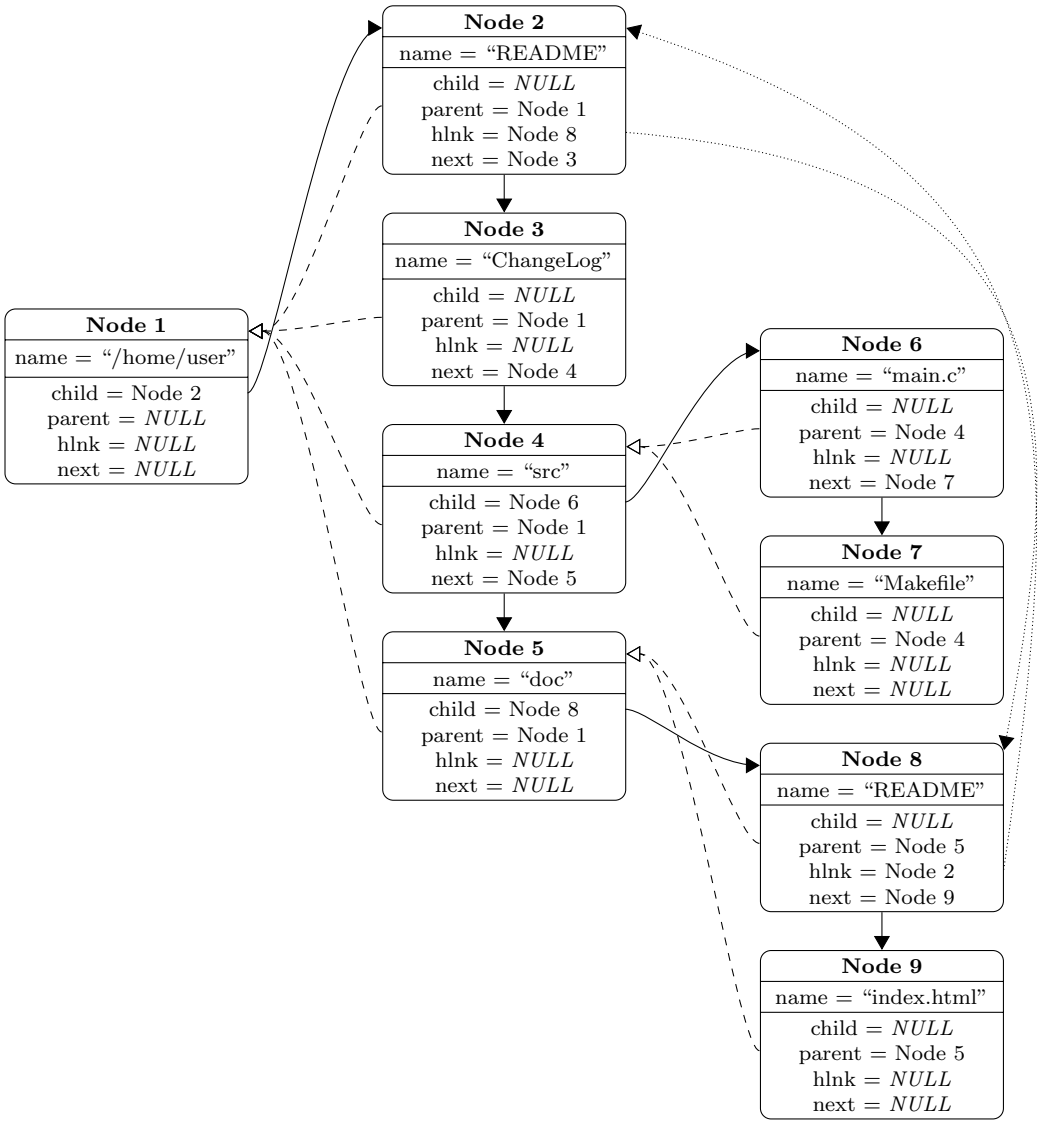

Figure 3.5. A directory structure implemented using a linked list

4 Storing linked lists in memory

As seen in chapter 3, a linked list consists of many small nodes, each of which is usually smaller than 100 bytes. As compression algorithms work better for larger amounts of data, it makes little sense to simply compress each node individually. It is therefore important to group the nodes together in larger blocks of memory on which compression can be applied. These blocks will be managed internally by the library, the application should be able to access each node without knowing about how it is stored in the blocks, as shown in figure 4.1.

Two methods of grouping the nodes will be researched: Grouping the *nodes* together in blocks of memory, or grouping its *elements* together.

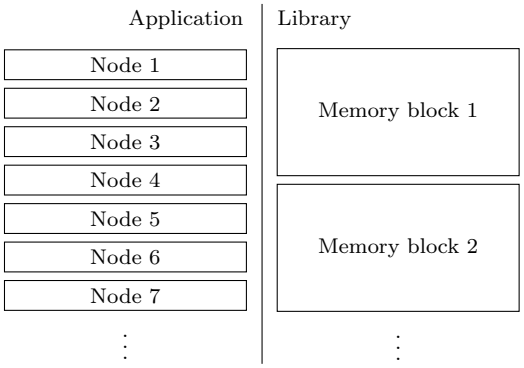

Figure 4.1. Schematical overview of the seperation of nodes and memory blocks

4.1 Grouping nodes

When grouping the nodes together, each node is considered as a single, small, block of memory, of which the contents are the responsibility of the application. The library does not have to know about the elements each node contains — it will simply store several nodes together in a larger block of memory. In this sense, the library is an alternative to the C memory management functions like *malloc* and *free*, which are traditionally used to implement linked lists.

Storage

The most obvious method to store multiple nodes in the same block of memory is to store them sequentially, without any padding or meta data in between. While this is certainly fast and has absolutely no memory overhead, it has several flaws.

One of these flaws is that some data structures have to be *aligned* to a certain memory address in order to improve portability and performance[Ahn05]. Some processors can only perform integer operations on integers which are located at a memory address that is dividable by the size of the integer. For example, these processors can not read a 32 bits (4 bytes) integer if it is located at an address not dividable by 4. While most modern processors can properly work with mis-aligned data, it does often still come at a performance cost. To avoid this, it may be necessary to add a few bytes of padding between each node, such that the next node starts at an aligned memory address.

With linked lists, it is common practice for nodes to be dynamically removed or added, and the memory 'freed' by removing a node could be re-used later on for a node that is added. This, however, is only possible if the library keeps track of which nodes are in use and which are free, which in turn is only possible if the library knows where in the block of memory each node starts and ends, which is complicated by the fact that the nodes are of a variable size.

To do this, we have to accompany each node with an additional field indicating the size of the node[Lea00]. When the size is known, the library knows where each node starts and ends, and will be able to iterate through the nodes. In addition to the size, we also have to indicate whether the node is in use or whether it has been freed by the application. Luckily, this is only one bit and can be stored in the same field as the size — assuming a 16 bits size field and that the node size does not exceed 32kB.

While it is now possible to find unused nodes and re-use memory when additional memory for a new node is requested, it is not efficient to iterate through the entire memory block to find out that only the last node is free. This can be improved by constructing a doubly linked list of free nodes within the (otherwise unused) memory[Lea00]. The references can be encoded as the offset of the linked node from the start of the memory block, this has the advantage that the references are small (a 16 bits integer is enough for a memory block up to 64kB) and that the references are independent of the address of the memory block — which can change after being compressed or decompressed.

A visualization of an example memory block is given in figure 4.2. The first two bytes are used for the reference to the first unused node, this is the *head* of the doubly linked list and provides a fast method to look for unused nodes: if it is *NULL*, the memory block is entirely filled up, if it is not, the free nodes can be traversed by going through the list. This field is followed by 4 bytes padding, which is necessary to make sure that the following node data starts at an 8-byte aligned address. For compression efficiency, this padding may consist of bytes containing zeros.

0 – 1	reference to first unused node
2 – 5	padding
6 – 7	status (*in use*) and size (*6 bytes*) of next node
8 – 13	node data
14 – 15	status (*unused*) and size (*14 bytes*) of next node
16 – 17	reference to previous unused node
18 – 19	reference to next unused node
20 – 29	padding
30 – 31	status (*in use*) and size (*20 bytes*) of next node
32 – 51	node data
52 – 53	padding
⋮	
4095	end of memory block

Figure 4.2. Structure of a 4kB sized block with 8-byte alignment

Algorithms

The size field before each node can be considered a reference to the next node in the memory block, essentially forming a singly linked list. The position of the next node can be calculated by adding the size and offset of the current node, taking into account that the size field itself is not included in the node size and that the node data is aligned to a dividable address. The formula is given in formula 4.1, where 'a' is the number of bytes alignment and the offsets refer to byte offset of the size field before the node, relative to the start of the memory block.

$$\text{next\_offset} = 2 + \text{current\_offset} + \text{size} + ((\text{a} - (\text{size} + 2) \bmod \text{a}) \bmod \text{a}) \quad (4.1)$$

Allocating a new node is be done by looking for the first unused node large enough to hold the new node by iterating over the unused nodes list. If a sufficiently large node has been found, the found node has to be removed from the unused node list by updating the previous and next references of the nearing unused nodes. If the unused node is large enough to hold another node, the reserved memory can be split up and a new unused node is then inserted after the recently allocated node, so that this space can be used again for a subsequent allocation.

This technique is called the *first-fit* approach[MR08]: the first node that is large enough to hold the new node is used. However, it is also possible that there is another unused node in the same memory block that is smaller than the first found node, but is still large enough to hold the new node. By using this node instead, less space is wasted on padding and the chance that there will only be small nodes available after a while — caused by splitting the larger nodes into smaller ones after a too large node has been used — is decreased. This approach is called *best-fit*, and causes less fragmentation at the cost of slightly more CPU cycles. This approach is easiest to implement by keeping the unused node list ordered by size, so that the smaller nodes are listed first and the larger nodes last. The first unused node in the list that is large enough to hold the new node is then automatically the best possible fit.

Deallocating a node can be as simple as setting the *unused* flag and adding it to the unused node list. However, when one or both nearing nodes are also unused, it is possible to combine these nodes into one larger node, so that subsequent allocations are more likely to fit, possibly decreasing fragmentation by keeping the amount of (too) small unused nodes at a minimum. Another advantage of combining multiple unused nodes into one is that the length of the unused node list is decreased, making allocations faster because less iterations are necessary to find a fitting unused node.

Compression effectiveness

In order to get an initial idea on how much the memory can be saved by using the above method, 'ncdu' has been modified to fill up one memory block of a configurable size with nodes and print out statistics. The tests were run on two directories: one directory with many small files with short names, and a directory with many large files with long names. For the block sizes 4, 8, 16 and 32kB were used, all with 8-byte alignment.

As seen in the previous paragraphs, not all memory in the memory block can be used for the node data, there is a little overhead caused by size information (metadata), padding for alignment, and the last few bytes of the block can't always be used because there is too little space left to hold another node. This "wasted" space has been measured in the tests and the results are displayed in table 4.1. All the nodes in the small files directory happened to be exactly 92 bytes, so the only padding that was necessary in that case was

the 6 bytes to make sure the first node is aligned. The nodes in the large files directory were of a variable size and more bytes of padding were necessary in that case. On average, the memory overhead is about 5% of each block.

directory	block size	#nodes	node size	metadata	alignment	unused
Small files	4096	42	3948	84	6	58
	8192	85	7990	170	6	26
	16384	170	15980	340	6	58
	32768	341	32054	682	6	26
Large files	4096	38	3896	76	122	2
	8192	75	7676	150	284	82
	16384	149	15482	298	546	58
	32768	289	31137	578	1053	0

Table 4.1. Measured memory overhead from grouping nodes in a memory block

To get an idea on how well compression works on these blocks, four compression libraries were tested: zlib, LZO, bzip2 and LZMA.

Zlib has three configuration parameters, each having an influence on the performance, memory usage and compression ratio. The memory usage depends on two parameters: the size of the history buffer and the size of the internal compression state. As it is impossible to take all combinations into account in the tests, 32kB was chosen for both the history buffer and internal state after a little experimentation with these options. The third parameter is the "compression level", which does not affect the memory usage but does have a significant effect on the speed and compression ratio, therefore all its nine possible options are included in the test result.

LZO has four separate functions for compression, each with different memory usage and performance.

While bzip2 also offers a compression level parameter like zlib, it did not seem to have any effect on either the performance or compression ratio for memory blocks of this size. Therefore only the lowest compression level, which uses the least amount of memory, is included in the tests.

The LZMA library also provides a compression level parameter, which does have a significant effect on both the memory usage and performance. Although this parameter has nine options, only two are included in the tests because both the memory usage and the speed degraded exponentially with each level, while the compression ratio only differed less than tenths of a percent.

The results of the compression are listed in table 4.2. These measurements confirm that better compression ratios come at the cost of slower compression and decompression times, as seen in chapter 2. It can also be seen that the compression ratio highly depends on the data being compressed: The small files directory can be compressed up to 4.62% (32kB block size with lzma), whereas the large files directory can't be reduced further than 17.94%. The effects of larger block sizes is also clearly visible: larger blocks provide better compression, and the (de)compression speed degrades almost linearly for all methods.

The measured memory usage for (de)compression was constant for all tests and is given in table 4.3. It is interesting to note that the memory usage for the LZO functions is twice that of the size advertised in the documentation. This can be explained by looking at the implementation: the actual allocated memory is calculated as a factor of the function **sizeof(unsigned char** $*$), which is 8 bytes on the (64bit) test system. The documentation likely assumed a 32bit architecture. bzip2 and LZMA both require more than a megabyte of memory for compression, which may limit their usefulness on smaller linked lists, where

Small files directory

	4kB		8kB		16kB		32kB	
method	ratio	speed	ratio	speed	ratio	speed	ratio	speed
lzo1x_1	19.73	72+ 21	19.09	98+ 38	18.68	135+ 69	18.47	216+ 161
lzo1x_1_11	19.70	33+ 16	19.09	54+ 33	18.69	88+ 71	18.46	158+ 142
lzo1x_1_12	19.73	34+ 17	19.09	42+ 33	18.67	85+ 64	18.46	150+ 141
lzo1x_1_15	19.73	66+ 16	19.09	86+ 33	18.68	147+ 63	18.47	203+ 133
zlib (1)	12.26	160+ 64	10.96	224+ 88	10.51	442+144	9.96	766+ 229
zlib (2)	11.96	148+ 45	10.91	180+ 68	10.47	414+122	9.95	833+ 211
zlib (3)	11.89	122+ 42	10.88	202+ 67	10.44	478+122	9.93	928+ 210
zlib (4)	11.96	171+ 47	10.57	291+ 71	10.13	853+132	9.63	1260+ 240
zlib (5)	11.43	179+ 41	10.28	316+ 65	10.16	723+121	9.47	1512+ 208
zlib (6)	11.47	200+ 41	10.20	377+ 66	10.00	898+119	9.47	1761+ 208
zlib (7)	11.43	218+ 40	10.28	473+ 67	10.00	1101+120	9.48	2095+ 207
zlib (8)	11.43	251+ 41	10.28	583+ 65	10.00	2041+118	9.48	3113+ 206
zlib (9)	11.43	302+ 41	10.28	674+ 66	10.00	3654+120	9.48	3250+ 206
bzip2 (1)	9.03	1413+180	7.59	2369+314	7.78	5084+578	6.05	7841+1093
lzma (1)	8.30	586+114	6.37	908+166	5.54	1412+267	4.62	2514+ 450
lzma (2)	8.30	2257+ 94	6.37	2581+145	5.54	3244+250	4.62	4467+ 432

Large files directory

	4kB		8kB		16kB		32kB	
method	ratio	speed	ratio	speed	ratio	speed	ratio	speed
lzo1x_1	38.45	151+ 25	36.52	192+ 48	36.06	294+ 93	31.56	454+ 178
lzo1x_1_11	38.53	64+ 22	37.50	115+ 44	36.82	228+ 93	32.23	411+ 160
lzo1x_1_12	38.38	65+ 21	36.67	110+ 42	36.24	215+ 85	32.04	387+ 159
lzo1x_1_15	38.43	236+ 19	36.54	272+ 40	36.02	355+ 78	31.52	516+ 162
zlib (1)	30.79	272+ 95	29.14	420+142	29.00	776+239	25.41	1489+ 414
zlib (2)	29.32	223+ 71	27.67	374+115	27.26	978+208	24.23	1500+ 377
zlib (3)	28.69	246+ 69	26.68	413+110	26.21	876+198	23.47	1717+ 367
zlib (4)	28.20	287+ 72	26.44	707+119	26.01	1055+210	23.03	2137+ 384
zlib (5)	27.86	332+ 67	25.93	599+110	25.38	1364+193	22.45	2342+ 357
zlib (6)	27.25	411+ 65	25.40	764+103	24.84	1603+183	22.07	3046+ 345
zlib (7)	27.12	752+ 68	25.26	982+103	24.46	2029+184	21.87	3850+ 341
zlib (8)	27.00	1268+ 65	24.98	2311+104	24.07	4624+180	21.69	8646+ 344
zlib (9)	26.81	1799+ 66	24.57	4375+108	23.79	9954+180	21.51	20991+ 334
bzip2 (1)	26.20	1888+318	22.55	3246+552	20.18	5859+995	18.69	9118+1881
lzma (1)	24.85	932+250	22.78	1524+425	21.91	2768+794	18.01	4642+1285
lzma (2)	24.78	2671+228	22.64	3363+421	21.80	4715+765	17.94	6834+1276

Table 4.2. Compression effectiveness and speed when grouping nodes. The compression ratio is given in percents of the compressed size compared to the uncompressed size, smaller is better. The speed is measured in µsec, once for the compression time and once for the decompression time.

the total memory usage for the uncompressed linked list is smaller than that required to perform the compression.

method	compression	decompression
lzo1x_1	131072	0
lzo1x_1_11	16384	0
lzo1x_1_12	32768	0
lzo1x_1_15	262144	0
zlib	71472	7152
bzip2	1118052	464144
lzma (1)	1413599	98424
lzma (2)	4985311	557176

Table 4.3. Measured memory usage for (de)compression

4.2 Grouping elements

Another method of storing linked lists in blocks of memory is by grouping the elements of the nodes together. This way, each node is internally split up among several memory blocks.

Storage

The large majority of all elements of a node in a linked lists have a fixed size. References to other nodes, for example, are fixed-sized pointers. Other information is commonly expressed in integers with a size in the range of 1 and 8 bytes.

Storing such elements together in one block of memory can be done easily and efficiently by storing the values sequentially. The size of each element is fixed, so we do not have to worry about storing size information with the data. Each element within a block can be accessed by an *index*: the byte offset within the memory block can be calculated by multiplying the index with the size of each element. This is similar to the concept of an array within the C programming language. With this method, there is also no need to worry about the alignment of the data, as all addresses are automatically dividable by the size of each element.

When grouping nodes together in blocks of memory, each node can be found when you know in which block it is stored and the byte offset of the node within that block. This is slightly more complicated when grouping the elements together, as the elements of a single node are divided into several blocks. To find a single element of a single node, you have to know in which block it is stored in addition to its index. It is, however, not very practical to have a separate identifier for each element of each node: in an application you have a certain (pointer to a) node, and you want to access one of the elements. Even when the elements of a node are grouped together in blocks of memory, you still want to be able to refer nodes, rather than to its elements.

The easiest and most efficient way to do this is by assigning a fixed number of elements to each memory block, and to (virtually) group memory blocks that hold the elements of the same node together. Take, for example, a linked list where each node has two elements, an integer of 1 byte and one of 8 bytes. When a node is added, the two elements are stored in two separate memory blocks and will have different memory addresses. By assigning the same virtual identifier to the two memory blocks and by keeping the index of the two elements the same, the node as a whole can be identified with only the block identifier and the index. This does mean that each memory block has to hold the same number of

elements in order for the index of the elements of the same node to be the same in all blocks. This means that the size of the memory block varies with the size of the elements it holds. Assuming we store 1024 elements in one block, the above example would use memory blocks of 1024 and 8192 bytes.

So far only fixed-size elements were discussed, but the library also needs to be able to handle variable-length elements. These are trickier to handle due to the fact that a variable-length element can not be found within a block by an index[1], but would require a byte offset instead, as seen in chapter 4.1.

In order to solve this problem, the same method as described in chapter 4.1 can still be used to store and compress the variable-length data, but with the addition of an extra memory block to hold fixed-size references to this data – consisting of a reference to a memory block and a byte offset – in the same way as a fixed-size element. In this sense, the variable-size element is replaced with a fixed-size "pointer", pointing to a chunk of memory managed by a memory allocator with compression support. As most variable-length elements in linked lists are strings (that is, arrays of integers with a size of 1 byte), extra padding bytes to have the data aligned to a certain memory address will not be required.

There is still one problem left unsolved: how can space of an unused node be re-used later on for a new node? The library needs some way to keep track of which nodes are in use, and which nodes can be freely used for new allocations. With all the elements of a node spread over several memory blocks, there is no single place in memory where general node information is stored, so there is no obvious place to store the node usage information. A solution is to store this information in a bit array: a special block large enough to hold one bit for each node within a block group, in which each bit indicates whether the node at that index is in use or not.

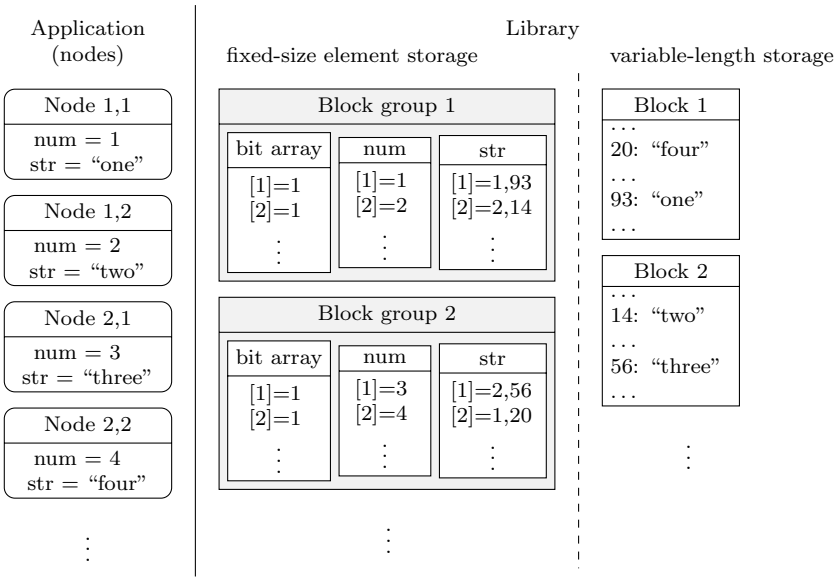

Figure 4.3. Example of how data is grouped into memory blocks

An example with four nodes is given in figure 4.3. Each node has two elements: a

[1]Actually, it can. But this would involve looping through all the elements within that block, which is not likely to be fast enough to be useful in any application.

number and a string. The four nodes are divided into two block groups, the first two are stored in group 1, the last two in group 2. The groups contain 2 memory blocks, each consisting of a list of the same element, of which each element is accessible with an index. The bit array indicates which indexes are in use and which are free. The variable-length string data is stored in separate blocks, these blocks are referred to from the fixed-size "str" block using the block number and byte offset where the data can be found.

Algorithms

Because all elements stored in the memory blocks have a fixed size, adding a new node is as simple as finding the first unused node by looking through the bit arrays, or creating a new block group in case all the others are fully in use. To improve performance, it may be wise to keep track of the number of free nodes within a block group, so that only the bit array of a group with free nodes has to be searched.

For variable-length elements, the same system as described in chapter 4.1 is used, and the same algorithms can be applied here.

To remove a node, the 'used' bit in the bit array has to be set to zero, and if the node has variable-sized elements, that data has to be marked as free as well. In order to improve the compression ratio for blocks that consist for a large part of unused data, the data of the elements itself can be reset to contain 'zero' bytes.

Compression effectiveness

To be able to make a reliable comparison in terms of compression effectiveness with the method of grouping nodes together, 'ncdu' has again been modified to perform similar tests with storing the elements together. One block group of a configurable size is filled with node data, and the variable-length element is stored into blocks of 16kB with an alignment of 2 bytes. For the block group sizes, 512, 1024, 2048 and 4096 nodes were tested, resulting in block sizes between 512 (512 elements of 1 byte) bytes and 64kB (4096 elements of 8 bytes). The tests were performed on the same two directories.

Table 4.4 lists the measured node size and memory overhead introduced by grouping elements together. Overall, about 90% of the used memory is actually used for the nodes, with an overhead of about 10%. This is slightly more than when grouping nodes together due to the added bit array and the reference to the variable-length data. The memory blocks holding the variable-length fields were not entirely filled, either. For linked lists without variable-length elements, the overhead is expected to be significantly less.

directory	#nodes	node size	bit array	block group size	var-length size
Small files	512	29696	64	27136	16384
	1024	59392	128	54272	16384
	2048	118784	256	108544	16384
	4096	237568	512	217088	32768
Large files	512	37448	64	27136	16384
	1024	78237	128	54272	32768
	2048	155726	256	108544	65536
	4096	313433	512	217088	114688

Table 4.4. Measured memory overhead from grouping elements in blocks

The measurements of the effectiveness of the compression algorithms are listed in table 4.5. Unlike when grouping the nodes together, storing more nodes into a single node

group – resulting in larger block sizes – does not seem to have any significant effect on the compression ratio. In fact, it can be seen that larger block sizes have a negative effect on the compression ratio with LZO and zlib for the small files directory.

Small files directory

method	512	1024	2048	4096
lzo1x_1	15.09	18.67	21.38	21.14
lzo1x_1_11	15.05	18.22	20.52	20.46
lzo1x_1_12	14.97	18.12	20.44	20.41
lzo1x_1_15	15.23	18.88	21.59	21.38
zlib (1)	7.85	9.41	10.25	10.06
zlib (2)	7.84	9.40	10.25	10.06
zlib (3)	7.83	9.39	10.24	10.06
zlib (4)	7.55	9.10	9.95	9.76
zlib (5)	7.53	9.04	9.86	9.66
zlib (6)	7.53	9.03	9.85	9.66
zlib (7)	7.53	9.03	9.85	9.65
zlib (8)	7.53	9.03	9.85	9.65
zlib (9)	7.53	9.03	9.85	9.65
bzip2 (1)	7.23	6.21	5.98	5.38
lzma (1)	4.77	4.29	4.13	3.62
lzma (2)	4.77	4.29	4.13	3.62

Large files directory

method	512	1024	2048	4096
lzo1x_1	31.36	31.57	31.00	31.19
lzo1x_1_11	30.12	30.47	30.00	30.13
lzo1x_1_12	29.99	30.40	29.93	30.06
lzo1x_1_15	31.77	31.86	31.22	31.37
zlib (1)	22.25	22.41	21.73	21.46
zlib (2)	22.16	22.26	21.62	21.43
zlib (3)	22.00	22.11	21.47	21.29
zlib (4)	21.38	21.54	20.87	20.66
zlib (5)	20.87	21.01	20.32	20.00
zlib (6)	20.86	21.04	20.31	19.99
zlib (7)	20.83	21.02	20.31	19.98
zlib (8)	20.82	21.00	20.28	19.97
zlib (9)	20.83	21.01	20.28	19.96
bzip2 (1)	22.58	22.02	20.34	19.59
lzma (1)	18.27	17.87	16.75	16.03
lzma (2)	18.27	17.86	16.75	16.03

Table 4.5. Compression effectiveness when grouping elements

The (de)compression speed has been omitted, as these measurements were performed on multiple memory blocks – one for each element and some for variable-length storage – while in practice only the blocks that are needed are (de)compressed. And as we have seen in chapter 4.1, (de)compression speed is mostly linear with the size of the memory block being compressed, which are roughly equivalent for both methods of grouping the nodes.

4.3 Comparison

In this chapter, the previously discussed methods of storing nodes will be compared with each other, and one method will be chosen for use in the library.

Compression effectiveness

The measurements of the compression ratios achieved with 'ncdu' have shown that the compression algorithms work better when grouping elements together. The difference, however, is only in the range of 1 to 5%. It is expected that grouping elements together works better with nodes with only fixed-size elements of which most have the same value, while grouping nodes together works better with variable-length elements.

Performance

In terms of performance both methods can be said to be equally fast: with both methods, creating a new node requires searching through a list to find unused memory. Although when grouping elements, this action has to be performed once more for each variable-length element. Removing a node is done by updating a few fields of meta-data and optionally clearing the unused data. Accessing the data is a matter of following references, no searching or looping is required with either method.

The compression and decompression speed is mostly linear with the size of the block being compressed, and with the memory blocks being of a configurable size with both methods, this performance overhead is also similar.

Grouping elements together has a theoretical performance advantage: in many applications, it is common to loop through a long list of nodes while only accessing a few elements rather than the entire node. When the nodes are grouped, the entire node will have to be uncompressed even if only one element is needed. By having less nodes in a memory block, more memory blocks will have to be accessed while looping through a list, possibly increasing the number of times a block has to be uncompressed or recompressed.

Implementation complexity

In order to handle variable-length elements when grouping elements together, the same method is required for grouping nodes together. As such, the implementation of grouping elements is more complex than when grouping nodes together. When grouping elements, there is also a greater overhead for storing and keeping track of meta-data, in order to group memory blocks together and hold information about whether a node is in use or not.

From the perspective of the application using the library, grouping elements may also be more complex because the elements have to be accessed individually and copying a node is not as simple as copying a single region of memory. The application also has to tell the library which elements are needed and of what type they are, limiting the possibilities to the data types and structures the library supports. When grouping nodes together, the application has more control over how it uses the (compressed) memory, and is not limited to linked lists.

Conclusion

In the end, the decision to use one method or the other is highly dependant on the application. For nodes with only fixed-size elements or for applications that mostly do not access all of its nodes' elements, grouping elements may be the best choice. For applications with variable-length fields or those that often access all elements, grouping nodes may be more efficient instead. To make a truly informed decision on one method or the other for a certain application in a certain situation, both methods should be implemented and thoroughly tested. This, however, takes a lot of time and there may still be situations in which the other method may have been a better choice.

In the case of 'ncdu', on which the tests and measurements were done, the simplicity of the implementation of grouping nodes greatly outweighs the slightly improved compression gained with grouping elements. Whether the theoretical performance improvement of grouping elements outweighs the complexity and overhead of that method can not be said at this point, but it is expected that both methods will provide acceptable performance with regular use of the program.

For this reason, and because an implementation of grouping elements will still require an implementation of grouping nodes, the current focus of the library will be using the method of grouping nodes together. For other applications and more detailed tests, the option of grouping elements will stay open for future work.

5 Memory Block Replacement

Now that there is a decision on how to store many nodes into several larger blocks of memory suitable for compression, there is still another problem to be solved: when, exactly, should a memory block be compressed or uncompressed? As the purpose of the library is to reduce the memory usage of an application, not all of the blocks can remain uncompressed in memory, but accessing the nodes within a block that is compressed is not possible either. Decompressing and later recompressing a block each time it is referenced is not going to be very efficient, either; compression takes a significant amount of time and should be avoided as much as possible.

So the best way to handle this is to have a fixed number of uncompressed blocks in memory, while all other blocks are stored in memory in a compressed state. When a node is requested that is not available in one of the uncompressed blocks, the block holding the node will be decompressed and its contents will replace one of the uncompressed blocks. The block that is replaced is then again stored in a compressed state somewhere in memory. The main question is now: *which* uncompressed block should be replaced when this happens?

This block replacement technique is very similar to the concept of *demand paging* found in modern CPUs and operating systems. As such, there has already been a significant amount of research on this topic, and there exists several answers to the previous question. This chapter discusses several commonly used block replacement algorithms with their advantages and disadvantages, in order to find a suitable algorithm for use in the library.

5.1 First-In, First-Out

Probably the most simple replacement algorithm is the First-In, First-Out (FIFO) algorithm. It works like a simple queue: when a new block is being inserted in queue (i.e. it is decompressed), the block that is least recently inserted will be removed from the queue.

An advantage of this algorithm is that it performs well for applications that only access the nodes in the same order as they were inserted into the (single dimension) linked list, and do not perform any modifications to the lists that could make the order of nodes in the list different from the order in which they are stored in the blocks. This algorithm will not perform optimally on more complex data structures like trees, and in practice it is common for the nodes within a linked list to be dynamically modified and logically re-ordered, making this algorithm unsuitable for most applications.

A more suitable algorithm based on FIFO exists, called *First-In Not-Used First-Out (FINUFO)* or *Second-Chance Replacement Algorithm* [Tan01]. This modification adds a single bit for each block which is set when a node within the block is accessed. When a block has to be removed from the queue, rather than simply removing the last block as with FIFO, the last block is only removed if this bit has not been set. If it has, the bit is cleared

and the block is re-added to be beginning of the queue, and the block that is then the last block in the queue will be considered using the same method.

This algorithm is an improvement over a plain FIFO queue and is still simple and efficient to implement, but it still has a major drawback: if all the blocks are accessed at least once before a compressed block has to be accessed – which is likely to be a common occurrence assuming an application often accesses nearing nodes in a short amount of time – this method will act as a FIFO again, with all its drawbacks.

5.2 Least Recently Used

As the name suggests, the Least Recently Used (RLU) algorithm replaces the block that has least recently been used. This involves keeping track of the time (or an another unit that increases with time, like a global instruction counter) that each block has been used last. When a new block has to be decompressed, the block that has not been used for the longest time will be compressed.

Various different implementations of this idea exist. Some implementations keep the time "backwards": instead keeping track of an absolute measurement of time, a relative measurement is used (e.g. "time elapsed since this block was last used"). This has the advantage of less storage space required to store the time field for each block, but requires this field to be constantly updated using a timer interrupt or a parallel process. Some implementations keep the list of blocks ordered by these times to make it faster to find out which block should be replaced, while for other implementations this sorting is too expensive and fully traversing the list of uncompressed blocks is more efficient.

While this algorithm comes with a significant implementation overhead compared to FIFO, it is generally better at making predictions about the activities of the application [McC05], and as such does a good job at keeping those blocks uncompressed in memory that are most likely going to be accessed again soon.

5.3 Least Frequently Used

The Least Frequently Used (LFU) algorithm is similar to LRU, but instead of keeping the time of the last access, the number of accesses ('frequency') is being kept instead. When a block has to be replaced, the block with the least number of accesses is removed.

LFU is similar to LRU in terms of implementation overhead and complexity, and can also do a good job at keeping those blocks uncompressed that are more often needed. However, LFU has one major drawback when compared to LRU: it reacts slowly to changes in the activity of the program. Suppose a program accesses a certain block many times in initialisation, the frequency counter of this block is then far higher than other blocks. But if that block is not accessed very often anymore after the initialisation is done, it will still remain uncompressed in memory for a significant time even if other blocks should then have priority. LRE can handle these situations better, as the number of times a block is accessed is irrelevant.

5.4 Dynamic- and working set replacement strategies

The previously described replacement strategies are known as "static replacement algorithms". Another type of algorithms exists, known as dynamic replacement algorithms.

These algorithms try to predict which memory blocks will be accessed in the near future and will load these in advance.

Working Set Replacement (WSR) algorithms are designed to handle a single shared memory region for multiple processes. Each process is assigned a "working set", which holds all the blocks that the process has used. After a certain amount of time, all blocks that have not been used are removed from this working set. Unlike the previously mentioned replacement algorithms, WSR keeps a dynamic number of blocks in memory.

Both dynamic replacement and WSR are designed for caching systems which handle more than one process (such as Operating System Caches) and impose a significant implementation overhead in order to keep statistics. These algorithms are, however, not suited for use in the library, as a single instance of the library can only be used from a single process.

5.5 Conclusion

The FINUFO algorithm is simple to implement, has only a small overhead and can easily handle managing many blocks without degrading the performance. In the case of this library, however, not that many blocks are kept uncompressed in memory[1], and replacing a block – which involves decompressing a new block and optionally compressing the old block when the contents have changed – is far more expensive than keeping track of an additional field. As such, LRU is the most suitable algorithm to use in the library.

[1]The entire aim of this library is to reduce the memory usage of an application. With a block size of 512kB, 100 uncompressed blocks already means having more than 50MB of "wasted" memory, which beats the entire purpose of performing compression in the first place.

6 API Requirements and Design

In order for an application to be able to make use of the functionality provided by the library, this functionality will have to be exposed to the application using an Application Programming Interface (API). As the target application (ncdu) is written in the C programming language, it makes sense to define the API for the same language.

The API is exposed to the application code by including a header file. The name compll.h is used for this library, standing for "COMPressed Linked List". As the C programming language does not have namespaces or other object-oriented paradigms, all functions, types and macros will be prefixed with compll_ to avoid name clashes with other functions and types used within the application.

6.1 Initialization

Before an application can make use of functionality of the library, the library first needs to be configured. Based on the previous chapters, the following configuration parameters can be identified:

Block size The size of each memory block. This parameter affects the number of nodes that can be stored in a single block, the maximum size of a single node, the compression ratio and the overall performance. Useful values are in the range of 1kB to 512kB.

Alignment The number of bytes on which each node start address should be aligned to. Possible values are: 1 (no alignment), 2, 4, 8, or 16.

Uncompressed block count The number of memory blocks to keep uncompressed in memory for fast access to the data. This parameter is a trade-off between memory usage and performance.

Compression algorithm Various compression libraries exist and each have their own parameters and advantages and disadvantages. Rather than using one fixed compression method, a better approach is to allow the application to specify two callback functions which do the compression and decompression. This has the advantage that any compression algorithm with any parameters can be used. In addition, this will make the library more portable: An application usually uses a specific build system in which linking to other libraries is handled. For libraries, linking to other libraries is more difficult without forcing a specific build system upon the application, therefore it is best to make use of the build system that the application already uses.

The callback functions used for (de)compression are called whenever a memory block should be compressed or decompressed. The compression function should accept a pointer

to the uncompressed memory that should be compressed (the "source" data) in addition to the size of that data, and a buffer to which the compressed data should be written to (the destination buffer). The actual length of the compressed data should be returned. The function for decompression has the same arguments, except that the source is now the compressed data and the uncompressed data should be written to the destination. The size of the uncompressed data is known by the library, so this does not have to be returned. The function prototypes are defined in listing 6.1.

```
unsigned int compress_block(const unsigned char *src,
                                  unsigned int   src_len,
                                  unsigned char *dest,
                                  unsigned int   dest_len);

void decompress_block(const unsigned char *src,
                            unsigned int   src_len,
                            unsigned char *dest,
                            unsigned int   dest_len);
```

Listing 6.1. Function prototypes for compression and decompression

The destination buffer is guaranteed to be large enough to hold the resulting data. The size of the compressed data is not known in advance when compressing a block, so a safe length for the destination buffer is chosen to avoid a possible buffer overrun. The tested compression libraries all provide a similar interface for single-pass (de)compression, which eases the implementation of the above-mentioned functions in the application.

The parameters can be initialized by calling compll_init(). This function accepts an argument for each parameter and returns 0 on success or a positive number on error. The complete function prototypes and type definitions required for initializing the library are defined in listing 6.2.

```
/* these types provide pointers to the (de)compression functions */
typedef unsigned int (*compll_compress_callback)(
   const unsigned char *, unsigned int,
         unsigned char *, unsigned int);
typedef void (*compll_decompress_callback)(
   const unsigned char *, unsigned int,
         unsigned char *, unsigned int);

/* the initialization function */
int compll_init(unsigned int    block_size,
                unsigned short alignment,
                unsigned short uncomp_count,
                compll_compress_callback   compress_cb,
                compll_decompress_callback decompress_cb);
```

Listing 6.2. Function and type definitions for initialization

6.2 Node manipulation

After the library has been initialized, it can make use of the compressed memory it provides. This section explains how this functionality can be used.

Identifying nodes

With traditional linked lists implemented using malloc(), a node is identified with a regular memory pointer. As the nodes stored in the library do not have a fixed memory address – they change each time the block is being compressed or uncompressed – regular pointers will not work. Instead, we have to define our own type.

As seen in chapter 4, a node can be identified with a block identifier and a byte offset. When all the memory blocks are internally managed in an array, the most efficient way to identify a block is with an index for this array. The size of the memory blocks is configurable up to at least 512kB, so the byte offset part of the node identifier should be at least 19 bits. If the node identifier is stored in a 32 bits integer, this would leave 13 bits for the block index, allowing only 8192 blocks. This is not enough for large linked lists, especially when a smaller block size is used.

As such, node identifiers are stored in 64 bits integers. This leaves enough room for a 24 bits byte offset – allowing block sizes up to 16MB – and a 24 bits block index, allowing up to 16.7 million memory blocks. The remaining 16 bits could be used for other information in the future.

Allocation and deallocation

In similar spirit to the traditional malloc() and free() functions, the library provides two functions for creating and removing nodes. The function prototypes are listed in listing 6.3.

```
/* the data type to identify a node, a 64 bits integer */
typedef unsigned long long compll_t;

/* create a node of a specified size (in bytes) */
compll_t compll_alloc(unsigned int size);

/* remove a node */
void compll_free(compll_t node);
```

Listing 6.3. Function and type definitions for (de)allocating nodes

Note that, because unused memory is reset with bytes containing zeroes to improve compression, newly allocated nodes are always initialized to zero. In that sense, compll_alloc() is similar to the built-in C function calloc().

If the requested node is too large to fit in a memory block or if the system is out of memory, the special value '0' is returned on compll_alloc() to indicate an error.

Accessing data

Accessing the data of a node involves resolving the abstract node identifier to a memory pointer, optionally decompressing a memory block if it is not in an uncompressed state already. The library also has to know whether the application is performing a read or a write operation on the node data: if a memory block has been modified while being in an uncompressed state, it will have to be recompressed when it is removed from the uncompressed blocks. If the memory block has not been modified, this compression stage is not necessary as the unmodified block is already stored in a compressed state.

This brings us to the function prototypes in listing 6.4. Note that the returned memory pointer is only valid until the compll_read() or compll_write() functions are called with another

node identifier or when compll_alloc() or compll_free() are called. For this reason, it is recommended to call compll_read() or compll_write() each time that an element in the node should be accessed – this function should be relatively fast if the node is already stored uncompressed in memory.

```
/* returns a pointer to the node data for reading */
const void *compll_read(compll_t node);

/* the same, for writing */
void *compll_write(compll_t node);
```

Listing 6.4. Function prototypes for accessing nodes

If the node can not be found (i.e. invalid identifier), the functions return a *NULL* pointer to indicate an error.

6.3 Debugging

Incorrect handling of pointers and dynamically allocated memory are a major source of application bugs. Memory leaks are caused by not calling free() on memory that can not be accessed anymore, and accessing data outside of the allocated memory range is also a common mistake. Various tools exist to aid developers to identify and solve problems like these, but these all assume that the traditional memory management functions like malloc() are used. Finding a memory handling bug in an application that uses the library will be a lot more difficult because such tools will not be able to do their job.

To make debugging easier, it is possible to replace the above library functions and types with their traditional uncompressed equivalents. Listing 6.5 shows how the previously described functions can be implemented with macros. To make switching between these debugging functions and the actual library easier, these replacement macros are included in compll.h and can be enabled by defining the macro COMPLL_NOLIB before including the header file.

```
/* we can use regular pointers to identify nodes */
typedef void * compll_t;

/* nothing has to be initialized,
   so the init function is replaced with a no-op */
#define compll_init(bs, al, cnt, comp, uncomp) {}

/* allocating a new node is equivalent to calloc() */
#define compll_alloc(size) calloc(size, 1)
/* ...and freeing a node is quite simple */
#define compll_free(p) free(p)

/* nothing has to be done when accessing nodes,
   as we already work with regular memory pointers */
#define compll_read(p)  (p)
#define compll_write(p) (p)
```

Listing 6.5. Replacement macros to help debugging applications

7 Implementation and realization

This chapter explains some decisions made while implementing and realizing the library and documents how the library works.

7.1 Memory block layout

As described in chapter 4.1, multiple nodes are stored within a single memory block. However, the storage format as described in that chapter had a few flaws that would make it impossible to handle blocks larger than 64kB or nodes larger than 32kB, and merging with a previous node would not have been very efficient.

To address the first problem without having to increase the storage requirements for the meta data, the offsets to other nodes are divided by the configured alignment before they are stored in the memory block. Because the byte offset of a node within a memory block is always a multiple of the alignment, no information is lost.

The size of a node, however, is variable, and does not necessarily have to be multiple of the alignment. But because the library uses the size only to figure out where the next node starts, the stored size does not have to be the actual size of the data part of the node, but can also include the overhead of padding and meta data. When the stored size is defined as "the number of bytes before the next node starts", it will automatically be dividable by the alignment and the same division trick as for the offsets can be used.

This does mean that not all alignment options can be used for all block sizes. For example, a block size of 512kB would require a minimum alignment of 16 bytes: the size of a node is stored in 15 bits, making the maximum node size with 16 bytes alignment $2^{15} * 16 = 512$kB. The library automatically increases the alignment to what is minimally required if the alignment specified by the application is too small. This is no problem for the application: the specified alignment is a minimum, after all — it does not matter when an alignment of 8 bytes is used if the application requested 4 bytes, any node that is aligned to 8 bytes is always aligned to 4 bytes as well.

When a node is marked as unused (i.e. upon calling `compll_free()`), it will be merged with adjacent nodes if these are also marked as unused. In order to do this efficiently, the library has to know where the next and previous nodes start and whether they are marked as used or not. In the case of the next node this is very easy, as the size of a node can be used to calculate where the next node starts, but this is not true for the previous node.

A solution to this is used in Doug Lea's memory allocator[Lea00]: rather than storing the 'used bit' within the size information of the current node, this bit is stored in that of the next node. This means that the used bit in the current node indicates whether the previous

node is used or not. If it is in use, there is no need to look further, as the node does not have to be merged. If it is not in use, the size of the previous node can be found in the last two bytes of its node data. From this, the offset of the previous node can be calculated.

Figure 7.1 gives a graphical overview of how the nodes are stored. The offset of a node — which is always aligned — points to the start of the data of the node. An unused node always contains three values in the data part: a link to the next and previous unused node (forming a doubly linked list that is ordered by the size of the nodes), and a duplicate of the node size at the end of the node data, which is used for a backwards merge. This means that the minimum size of a node is 6 bytes, in order to be able to store these fields when the node is freed.

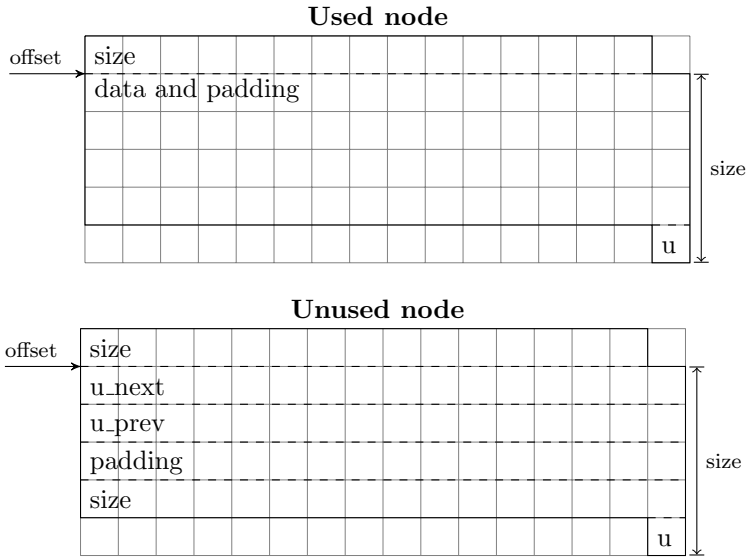

Figure 7.1. Node format. Each cell signifies one bit, a row contains 16 bits (two bytes).

7.2 Memory block management

The memory blocks are managed in two arrays: one array that holds information about all blocks that are present (the *ablocks* array) and a smaller array for the blocks that are being kept uncompressed in memory (the *ublocks* array). The definitions of these arrays are shown in listing 7.1.

The data field of the ablocks array contains a pointer to the compressed contents of the block, or NULL if the block is not compressed. The data_len field indicates the size of the compressed data, which is used when decompressing the block. If the block is available in an uncompressed state, ublock holds the index of this block in the ublocks array, otherwise this field is -1. The free field indicates the size of the largest free node within the block, and can be used to find a suitable block when a new node is allocated.

The ublocks array contains a pointer to the uncompressed data block (the size of which is always the configured block size), an index back to the ablocks array and a last access counter, which is updated each time the block is accessed.

Each memory block has a fixed position in the ablocks array, and the index to this array is used to uniquely identify a block. When a node within a compressed block is being

```
/* the ablocks array */
struct ablock {
  unsigned char *data;
  unsigned int   data_len;
  short          ublock;
  unsigned short free;
} *ablocks;

/* the ublocks array */
struct ublock {
  unsigned char *data;
  int            ablock;
  unsigned int   lasta;
} *ublocks;
```

Listing 7.1. Struct definitions of the block management arrays

accessed, the block is allocated a slot in the ublocks array and its contents are decompressed. The compressed data will remain in memory until the contents of the uncompressed block are modified, in which case the compressed data is thrown away and ablocks.data is set to NULL. If the block is later removed from ublocks (i.e. it is replaced with another block), the contents will only be compressed again if the compressed data has not been previously thrown away. This assures that compression is only performed when necessary and that memory that is not required anymore is freed as soon as possible.

7.3 Internal functions

The complex functionality of the library is internally divided into a number of functions that each perform a more simple task. Below is an explanation of what these functions do and how they were implemented.

int block_uslot() ±45 lines

This function allocates a slot in the ublocks array and returns the index to this item. If all items in the ublocks array are in use, the block with the lowest lasta value will be compressed if it has been modified and the uncompressed block will be re-used. The lasta field of the returned ublock is also updated.
Realising this function was fairly straight-forward and it worked the first time.

int block_alloc() ±40 lines

Creates and allocates an empty memory block. The index to the ablocks array is returned. This function makes sure that the new block is available in an uncompressed state by calling block_uslot().
While this function is also fairly straight-forward, it is slightly more complex as it has to deal with managing the variable-sized ablocks array, reallocating the array if it is not large enough.

int block_load(int block) ±25 lines

This function makes sure that the memory block (identified by its ablocks index) is available in an uncompressed state and that its lasta field is updated. The ublocks index is returned. If the block is not already present in the ublocks array, this function will call block_uslot() and decompress the block.

The first implementation of this function did not check whether the block index was valid, which resulted in invalid memory operations if the application contains a minor bug. The function now returns -1 on error.

void block_dirty(int block) ±5 lines

Mark the given block as 'dirty'. That is, delete the compressed data if it is in a compressed state. This makes sure that when the block is removed from the ublocks array, its contents will be compressed again.

Due to the small size and the fact that this function is called quite often, it is actually implemented as a macro.

int block_getfree(short size) ±45 lines

Find a memory block that has enough unused space to hold a node of the given size. Internally calls block_load(), so the block is available in an uncompressed state and its lasta field has been updated. If no suitable memory block could be found, this function creates a new one using block_alloc() and initializes the block with a single unused node that fills the entire block.

The complexity of this function lies in the operation of formatting a new block with one unused node. The offset and size of this node is calculated from the block size and alignment, and requires care to make sure that the node still fits in the memory block even if the block size is not a multiple of the alignment, and to make sure that the node data does not overlap with the head pointer of the unused node list.

void node_addfree(int block, int offset) ±30 lines

Adds the specified node to the correct position in the unused node linked list by updating the links in the unused node data. ablocks.free is also updated if this node is larger than the previously largest unused node.

Performing operations on doubly linked lists is a complex task even with normal linked lists. This is complicated by the fact that the references in this case are encoded offsets within the memory block. These offsets first have to be decoded to memory pointers in order to read them, and later encoded back to offsets when writing to the block.

void node_rmfree(int block, int offset) ±20 lines

Removes the specified node from the unused node linked list and updates ablocks.free if the removed node was the largest in the memory block.

Removing a node from a doubly linked list is generally easier than adding one, but the same encoding and decoding as with node_addfree() was used so realising this function was still not very easy.

int node_alloc(int block, short size) ±35 lines

Allocates a node of the given size within the memory block. This is done in the following steps:

1. Walk through the unused node linked list and find the first node that is larger than or equal to the given size.
2. Remove the node from the unused nodes linked list (node_rmfree()) and update its used flag.
3. If the found unused node is large enough to hold another node, the node is split and the second node marked as unused and added to the unused node linked list (using node_addfree()).

The first two steps were straight-forward and easy to implement. The last step, however, was more complex and error-prone as there are various factors that determine when a node is 'large enough'. This was again complicated by the decoding and encoding of the offset and size values within the block.

Using the above functions, the library functions could be trivially implemented as follows:

compll_t compll_alloc(unsigned int size) ±30 lines

1. Check that the requested node size is not too large to fit within a block
2. Increase the requested size with 2 (to hold the meta data for the next node) and align the size to the configured alignment.
3. Find or create a suitable memory block using block_getfree().
4. Allocate the node within that block using node_alloc().
5. Call block_dirty().

void compll_free(compll_t node) ±35 lines

1. Make sure the correct block is available uncompressed using block_load().
2. Mark the node as unused and fill its data with zeroes.
3. If the above node is unused, remove the above node from the unused linked list using node_rmfree() and merge the two nodes into one.
4. Do the same with the node below.
5. Add the resulting node to the unused linked list using node_addfree().
6. Call block_dirty().

const void *compll_read(compll_t node) ±10 lines
Call block_load() and return the pointer to the location of the node within the uncompressed block.

void *compll_write(compll_t node) ±10 lines
Same as compll_read(), but also call block_dirty().

8 Test Results

Now that the library has been implemented, there has to be some indication of whether it works according to our specifications. To this end, some tests have been performed, which will be explained in this chapter.

8.1 Test setup

The tests are performed on ncdu and are designed to simulate regular browsing behaviour, in order to get a feel on whether the performance degradation is noticeable in normal — or at least practical — use of the application. The following operations are tested:

scan Upon starting ncdu, it will recursively read the directories and fill the data structures with information about the files and directories it has encountered. The directory used for testing consists of approximately 100 subdirectories and 225.000 files, with each file and directory being internally represented by a node.

open Each time a new directory is opened while browsing through the (in-memory) directory structure, two operations have to be performed: the entire directory listing has to be sorted by the size of each item, and the directory has to be scanned once to find out which item is to be selected in the browser interface. These operations are usually quite fast for smaller directories, but in practice large directories also do exist. To make sure that the browsing experience of ncdu stays acceptable the 'open' test is performed on a large directory containing 25.000 items.

browse The browser interface of ncdu is used to display and scroll through a directory listing. This interface is re-drawn every time the user presses a key or scrolls through the list, and as such the speed of this operation has a significant effect on the responsiveness of the application. In the tests, the browser window will be drawn a hundred times, each time scrolling down one item in the previously opened directory of 25.000 items.

delete ncdu has the ability to delete directories from within the browser interface. The performance of this feature is also tested by (virtually) deleting the entire directory structure. The actual file deletion calls are commented out to make sure that the files are not really deleted from the filesystem, but only to make ncdu think that they are. This operation involves recursively scanning through the in-memory directories and removing items as they are being deleted.

For each test, the CPU time and memory usage are measured. The CPU time is the time that the CPU was busy executing the application instructions, and is also called *user time*

on UNIX systems. The measured memory usage is taken from the system implementation of malloc() and represents the total amount of dynamically allocated memory that is in use at that point. The CPU time is measured for each of the above operations, the memory usage is only measured once, after the directory has been scanned and is entirely stored in the application memory.

8.2 Measurements

The effects of three configuration variables on the performance and memory usage are measured by changing these variables within the tests. The tested variables are the number of uncompressed blocks to keep in memory, the block size, and the used compression technique. The effects of different alignment options are not tested, as ncdu requires at least 8 bytes alignment and this variable is not expected to have a significant effect on either the memory usage or the performance.

The effects from changing the number of uncompressed blocks are displayed in figure 8.1. It is interesting to note that the library has little effect on the performance of the 'browse' test. This can be explained with the fact that this test only accesses a (relatively) small number of nodes, which are possibly all available in the uncompressed blocks and thus no compression or decompression is being performed. The other operations are significantly slower with the use of the library, but there does not seem to be much of a performance difference with the different options for the number of uncompressed blocks. One exception to this is when this number is set to 100, in which case all nodes within the directory being opened likely fit into the uncompressed blocks. As expected, the memory usage increases as more uncompressed blocks stay in memory. A good compromise between memory usage and performance in this case would be to have 10 uncompressed blocks.

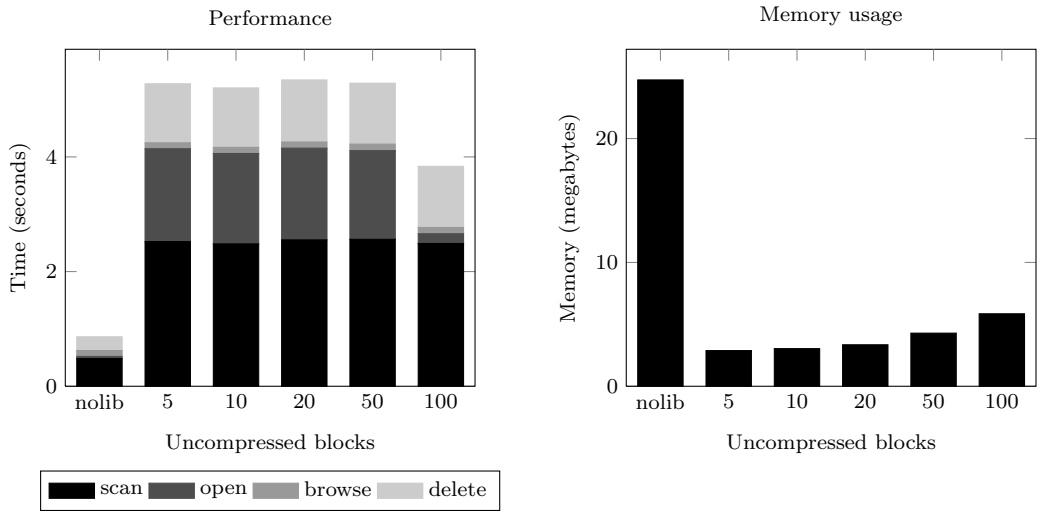

Figure 8.1. Effects of the number of uncompressed blocks on the performance and memory usage of ncdu. The block size is configured to 32kB and zlib-5 is used for compression.

Figure 8.2 shows the effects of using different block sizes. The first thing that can be noticed is that larger block sizes provide better performance with the 'scan' operation. This can be explained with the fact that, in the current implementation, finding a block with

enough free space can be slower than finding a fitting unused node within a block; making node allocation faster when there are less blocks and more nodes within a single block. It can also be seen that with 512kB blocks, the 'open' action performs significantly faster than with smaller blocks. This can again be explained with the idea that all the nodes for the directory fit within the uncompressed blocks.

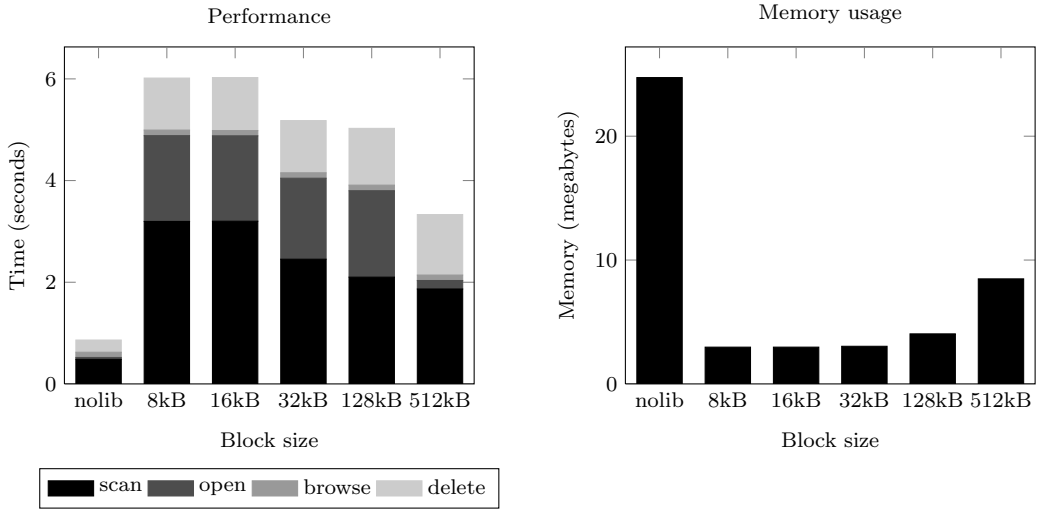

Figure 8.2. Effects of the block size on the performance and memory usage of ncdu. The number of uncompressed blocks is configured to 10 and zlib-5 is again used for compression.

The memory usage is constant with block sizes between 8kB and 32kB, and starts increasing after that. The number of uncompressed blocks is the same, so for larger block sizes, more memory is required to hold these uncompressed blocks. On the other hand, larger blocks compress better than small blocks, as seen in earlier tests in chapter 4. The point at which the improved compression ratio can not make up for memory used for the uncompressed blocks lies somewhere between 32kB and 128kB in these tests. A block size of 32kB is a good compromise.

The effects of using different compression algorithms can be seen in figure 8.3. Four compression algorithms are used in addition to a 'copy' algorithm, which simply copies the block contents as-is and does not really provide compression. One interesting effect that can be observed is that even without performing compression, the library still uses less memory than the system implementation of malloc(). This can be accounted to the fact that the library has less overhead per node; with the library, only two bytes of meta data are required per node, this is likely larger with the system implementation of malloc().

Similar to the results from chapter 4, LZO is the fastest algorithm but does not provide very strong compression, zlib is a good compromise between performance and compression and LZMA has the best compression but is relatively slow. bzip2 does not provide any advantage compared to the other algorithms; it is both slower and has weaker compression than LZMA.

The memory usage required to perform the compression and decompression is not included in the measurements as this memory has already been freed at the time that the memory usage is measured. The required memory usage for compression and decompression is constant for each algorithm and has already been measured before in table 4.3 in

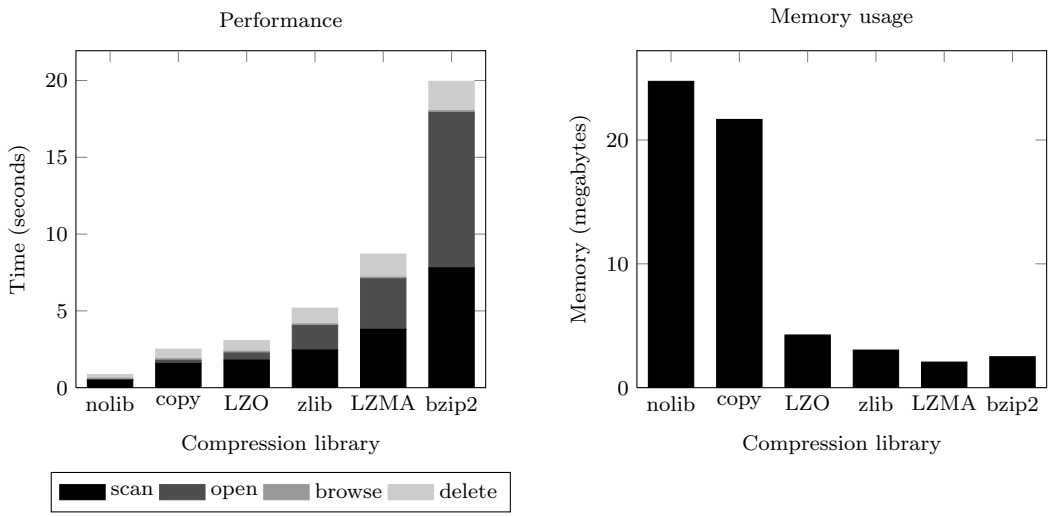

Figure 8.3. Effects of different compression libraries on the performance and memory usage of ncdu. A block size of 32kB and 10 uncompressed blocks are used for these tests.

chapter 4.1.

8.3 Conclusions

These primarily test results indicate that the use of the library slows down most operations with a factor 4 to 6, depending on the chosen configuration. The memory usage with use of the library is around 8 times lower than without, which is an overall compression ratio of about 20%. These results exceed the initial expectations and show that compression of application memory can be done without having a very large effect on the performance.

These tests were performed on an initial implementation of the library. It is expected that, as the library is being used over time, performance issues in the implementation or its algorithms will be isolated and improved, so that the library will perform even better in the future. Further optimizations can also be done within the application (in this case, ncdu), so that the algorithms are designed to make more use of the uncompressed blocks. For example, the 'open' operation within ncdu uses the merge sort algorithm to sort the list, switching to a more locality-aware algorithm could significantly improve the performance.

When focussing on the tested operations within ncdu, the operation that has most influence on the responsiveness of the interface (the 'browse' operation) is still fast even with the use of the library. Opening a large directory ('open') is significantly slower, but when taking into account that an average directory does not contain more than 100 items and directories as large as the one in this test are not that common in practice, a waiting time of one or two seconds is still acceptable. The 'scan' operation is only being performed at the startup of ncdu and takes in practice a lot more than a few seconds due to the slowness of hard drives (which was not considered in these tests). Three seconds for scanning a full directory structure containing 225.000 files is still quite fast. The 'delete' operation is similar: most filesystems are unable to actually delete that many files within a second, so the time added by the library is not likely to be too noticeable.

A good compromise between memory usage and performance can be obtained with a

block size of 32kB, 10 uncompressed blocks, and the zlib compression library.

9 Conclusions and Recommendations

This chapter looks back at the project, confirms whether the resulting library meets the earlier defined requirements and gives a few recommendations for further research.

9.1 Summary and Schedule

This project is divided into three stages: analysis, design and implementation. The analysis stage introduced the problem that this project should solve and explained more in detail how data compression works and what linked lists are and how they are used.

Three major design issues were handled and solved in the design stage. The first issue was how to store the nodes of a linked list into larger memory blocks. After comparing two possible solutions, it was decided to group nodes as a whole together rather than splitting their elements over several blocks. The second issue was about finding an optimal algorithm which decided when, and more importantly, which memory block should be compressed or decompressed through operation of the library. The Least Recently Used replacement strategy was found to be the most fitting for use in the library. Finally, an API was designed which exposes the functionality of the library to the application and also includes helpful macros for debugging.

The actual library was written and tested in the implementation stage. Some minor changes in the storage of the nodes within the memory blocks were made to improve performance and to handle larger block sizes. The library was tested by modifying ncdu to make use of the library and performing several tests on ncdu with different configuration values.

Figure 9.1 displays the time distribution of the above mentioned stages and makes a comparison with the estimated time distribution made in chapter 1. The actual time distribution is based on the activity log in appendix B.

9.2 Project results

Table 9.1 compares the results of the library with the initially specified requirements, and confirms that the implemented library confirms to our specification.

Both the large memory savings and little performance degradation from using the library exceed our initial expectations.

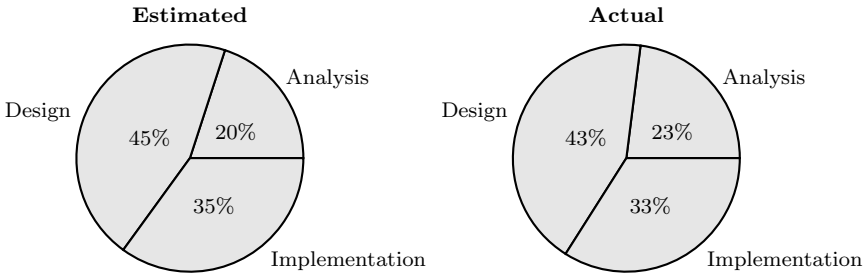

Figure 9.1. Estimated and actual time distribution among the three stages

Requirement	**Confirms**	**Explanation**
A. Any data structure	Yes	The library itself does not enforce any specific type of linked list. In fact, the library can even be used for non-linked list data stored in memory by using it as a replacement to the C *malloc()* function.
B. Nodes of a variable size	Yes	The size of each node is variable and should be specified at the time of allocating the node. The current implementation, however, does impose a maximum on the node size, which is slightly smaller than the configured block size.
C. Memory saving on large lists	Yes	Based on the test results, the memory usage that is saved by using the library is about 80%.
D. Only in application memory	Yes	The library internally uses the system implementation of *malloc()* to store all its data. No other forms of data storage are used.
E. Support for all basic operations	Yes	Using the library, nodes can be created using *compll_alloc()*, read using *compll_read()*, modified using *compll_write()*, and deleted with *compll_free()*.
F. Acceptable performance	Yes	As seen in the test results, most operations perform about 5 to 7 times slower with use of the library. This is in practice acceptable for ncdu.
G. Packaged and documented	Yes	The library is packaged together with a Makefile, small test suite, and a README file containing instructions on the usage of the library within the application. (Appendix C).

Table 9.1. Comparison of the implemented library with the specified requirements. The requirements are listed in the same order as in chapter 1.

9.3 Recommendations

- The option of grouping elements together in blocks of memory has been considered to be not worth the added complexity in this project. For other applications and for future research, however, it may still be worth to make a more in-depth comparison between the two methods in terms of performance, and/or allow both methods to be used within the library.
- The main focus of this project was on single-threaded applications. Using the library in multi-threaded applications would require locks around each call to the library, which is far from optimal. Further research can be done to improve the support and performance in multi-threading environments.
- The current implementation of the library does not support nodes that are larger than the configured block size minus overhead. However, it may be possible to modify the library to allocate such large nodes into separate blocks, so that large nodes can still be used.

Bibliography

[Ahn05] Song Ho Ahn. Data alignment. `http://www.songho.ca/misc/alignment/dataalign.html`, 2005.

[Ast04] Owen L. Astrachan. Huffman coding: A CS2 assignment. `http://www.cs.duke.edu/csed/poop/huff/info/`, February 2004.

[Bou98] Paul Bourke. Delta coding and run length encoding. `http://local.wasp.uwa.edu.au/~pbourke/dataformats/compress/`, May 1998.

[Lea00] Doung Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, April 2000.

[LH87] Debra A. Lelewer and Daniel S. Hirschberg. Data compression. *ACM Computing Surveys*, 19:261–296, 1987.

[McC05] Terrance McCraw. Virtual memory page replacement algorithms. Technical report, Milwaukee School of Engineering, February 2005.

[McF92] Andy McFadden. Hacking data compression. `http://www.fadden.com/techmisc/hdc/`, October 1992.

[MR08] Miguel Masmano and Ismael Ripoll. A constant-time dynamic storage allocator for real-time systems. *Real-Time Systems*, 40:995–1026, November 2008.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

Appendices

A Competences

	Competence	Explanation
1	Draw up a requirements report	Based on the final project proposal, I will, in a very early stage of the project, write a detailed problem specification which covers all the requirements to which the library should confirm.
2	Make choices	In the design stage of the project, I will research two methods to internally storing nodes in blocks of memory and will make an informed decision based on this research.
3	Formulate a project plan	Based on the detailed problem specification, I will, in a very early stage of the project, write a detailed solution analysis covering the project stages and subjects and a time schedule.
7	Monitor the process	In an early stage of the project I will create a global plan of action, and at the end of each week I will write down my past activities in a log, compare this log to the schedule, and adjust the schedule where necessary.
9	Deliver	At the end of the project I will deliver a detailed report explaining in detail that the library meets the earlier defined requirements. I will also give a presentation about this project outlining what I have done and demonstrating the result, and deliver a final professionally packaged library with well-documented code.
17	Reflection	By having a performance interview around the middle of the project period, I will receive feedback on my behavior, formulate my weaknesses in a short report, and try to improve them in the second half of the project. After which the final performance interview determines whether I succeeded in improving myself.
19	Flexibility	In order to prove that I can easily adjust myself to any working environment, I will work on the project from my home most of the time. Working in this environment requires discipline and perseverance because there is nobody close to me to motivate me to continue working on the project. I will also keep a log of my activities to ensure that I will not stray away from the project.
21	Relations management	By working at home, it is important to keep active correspondence with my company supervisor. I will keep him updated about the project progress at least once a week.
22	Responsibility	Whenever either the company supervisor or university supervisor ask for a reasoning behind my actions, or whenever I feel they should be informed about something, I will take responsibility and adequately explain myself and take the proper steps when necessary.
4	Design	I will design a library based on the requirements defined in an earlier stage of the project. The implementation will prove that the design does indeed confirm to the requirements.
5	Assemble	In the implementation stage of the project, I will assemble the ideas found in the design stage together into one product: the library.
12	Suggesting improvements	At the last stage of the project I will write a short report covering possible future research topics to improve the library.
20	Innovate	Due to the unique problem at hand, I will need to think of innovative technologies to come to a solution that confirms to the specified requirements.
24	Giving advice	At the last stage of the project I will write a short report with advice on situations in which the use of the library is (un)suitable, and suggest methods to improve the suitability of the library for certain situations.

B Activity log

Week	Activity
1	- 8 February, initial meeting with university supervisor. We spoke about the project description and structure. - Created initial report layout and structure. - Discussed final project description with the company supervisor over email. - Started on chapter 1.
2	- Formulated the competences (appendix A). - Started on chapter 2 - 18 February, second meeting with university supervisor. He gave feedback on the report layout and chapter 1. - Processed the received feedback and finalized chapter 1.
3	- Put together the Plan of Action, verified this with company supervisor (Monday) and delivered it at the University (Thursday). - Finished the initial version of chapter 2. - Started on chapter 3.
4	- Finished the initial version of chapter 3. - Started research on memory allocators and grouping nodes together in one memory block.
5	- Continued the work on grouping nodes together in blocks of memory. - Modified 'ncdu' to gather some memory and compression statistics.
6	- Processed, documented and explained the statistics gathered from 'ncdu'. - Asked the company supervisor for feedback on the current report (Wednesday) and discussed about the report (Thursday).
7	- Processed the feedback from the company supervisor. - Sent the current report to the university supervisor (Tuesday), and discussed about the report and the project with the company supervisor and university supervisor.
8	- Finished the chapter about grouping nodes in blocks of memory. - Made a start on grouping elements in blocks of memory.

Week	Activity
9	- Wrote assessment report and sent it to the university. - Modified 'ncdu' to gather statistics about grouping elements together. - Finished chapter 4.
10	- Researched demand paging algorithms and wrote chapter 5. - Defined an API and wrote chapter 6.
11	- Sent the current report to the company supervisor for feedback (Monday) and processed the feedback (Tuesday). - Sent the report the university supervisor (Wednesday), discussed about the report and project advancement (Thursday) and processed the feedback in the report (Friday). - Started on the implementation of the library.
12	- Finished the initial implementation of the library. - Started on a test suite to test whether the library works.
13	- Finished the small test suite and fixed some issues with the implementation of the library. - Wrote the chapter about the implementation.
14	- Implemented the library in ncdu. - Modified ncdu to perform performance and memory tests. - Started on the chapter about the test results.
15	- Finished the tests on ncdu. - Finished the chapter about the test results. - Wrote the last chapter (Conclusions and Recommendations).

C The README File

```
compll - A compressed memory allocator
=======================================
```

DESCRIPTION

compll is a small library that provides replacement functions for malloc() and free() that allow the application memory to be compressed.

INSTALLATION

Due to the small size of this library, it is recommended to simply add the compll.h and compll.c files to your project and compile and statically link them into your application as if they were part of your project.

The library should compile on any conforming C99 compiler, or a C89 compiler with support for 'long long' integers. [unsigned] short types are assumed to be exactly 16 bits wide. So far, only GCC has been tested.

A minimal Makefile is included to build an object file from the library and to run the tests. The test suite requires quite a bit of memory in addition to the 'prove' program, which should come with a default installation of Perl.

USAGE

Read the compll.h header file for the function prototypes and documentation on how to use these in your application. Refer to the report "Design and implementation of a compressed linked list library" for a more detailed explanation of the API and inner workings of the library.