Cache Oblivious Data Structures

by

Darin Ohashi

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2001

Abstract

This thesis discusses cache oblivious data structures. These are structures which have good caching characteristics without knowing Z, the size of the cache, or L, the length of a cache line. Since the structures do not require these details for good performance they are portable across caching systems. Another advantage of such structures is that the caching results hold for every level of cache within a multilevel cache. Two simple data structures are studied; the array used for binary search and the linear list. As well as being cache oblivious, the structures presented in this thesis are space efficient, requiring little additional storage.

We begin the discussion with a layout for a search tree within an array. This layout allows Searches to be performed in $O(\log n)$ time and in $O(\log_L n)$ (the optimal number) cache misses. An algorithm for building this layout from a sorted array in linear time is given. One use for this layout is a heap-like implementation of the priority queue. This structure allows Inserts, Heapifies and ExtractMaxes in $O(\log n)$ time and $O(\log_L n)$ cache misses. A priority queue using this layout can be built from an unsorted array in linear time. Besides the n spaces required to hold the data, this structure uses a constant amount of additional storage.

The cache oblivious linear list allows scans of the list taking $\Theta(n)$ time and incurring $\Theta(\frac{n}{L})$ (the optimal number) cache misses. The running time of insertions and deletions is not constant, however it is sub-polynomial. This structure requires ϵn additional storage, where ϵ is any constant greater than zero.

Acknowledgements

There are many people who helped me complete this thesis. Most importantly is Ian Munro, my thesis supervisor. We spent a lot time discussing this work and his ideas and suggestions were invaluable. Ian also helped turn my initial drafts into a thesis. Not an easy task, I assure you. I would like to thank Jonathan Buss and Naomi Nishimura, the readers of my thesis. Their comments helped me make this a far more readable document. They may have finally broken my love of the comma.

I would also like to thank my friends; Laura, Jenn and Jeremy. They helped me maintain a life outside of university which allowed me to remain happy and relaxed while at university. I would also like to thank Andrew and Chris, my friends and officemates. They helped create a non-hostile environment in which to work. Finally I would like to thank my parents who have supported me for as long as I can remember. I would never have made it this far without their support.

Contents

List of Figures

Chapter 1

Introduction

Moore's Law was a conjecture that the density of transistors in chips would double every twelve months. Although it was later revised to eighteen, the exponential growth has continued for the last 35 years. This has lead to processors whose speed increases at approximately the same rate, and today processors are available which run at over 1 Gigahertz. For memory technology, this increase in density has lead to chips capable of storing hundreds of Megabytes. However, the access times have not increased as quickly. Faster memory requires using different technologies which are expensive, in terms of space and cost.

This speed difference is a problem because every instruction and piece of data that the CPU uses is stored in main memory. If the CPU needs to access main memory for every instruction, it will spend many cycles waiting. An even worse situation occurs in systems which require more data than they have space in main memory. These systems usually use hard drives to store their data. Hard drives have worst case individual access times thousands if not millions of times slower than memory. If the CPU requires a piece of data stored on disk the data must be read from disk to memory and then from memory to the CPU. The obvious solution for this problem is to keep a copy of the most heavily used data in main memory. If most of the required data is in memory then disk accesses are dramatically reduced. This idea can also be applied to main memory by adding a small amount of fast memory between main memory and the CPU. These are examples of the most basic form of caching.

Caching allows blocks of data to be moved from a slow storage medium to a fast one. Any additional accesses into that block of data can be made directly from the faster location. If there are multiple accesses to a single block in the cache, then the cost of the transfer from slow memory to fast memory can be amortized over these additional accesses. Unfortunately there is no guarantee that more than a single element of the data block will be used. If the data in the cache is only accessed once, then the cache does not provide any advantages. Therefore, to fully exploit the benefits of the cache, algorithms must be designed with this notion explicitly in mind.

Many algorithms and data structures have been designed to work with the cache. The blocking approach of Golub and Van Loan leads to such an algorithm for matrix multiplication [GvL89]. The B-Tree [BM72, CLR91] is an example of a data structure that can be tuned to exhibit good caching characteristics. Most of the work in algorithms and data structures with good caching characteristics is cache aware. A ***cache aware*** algorithm (or data structure) is one which has knowledge of the caching architecture on which it is running. This knowledge is used to set parameters which allow the algorithm or data structure to exploit the cache.

Although aware solutions have good caching characteristics, they do have their downsides. Being aware means that the program must know some details about the system on which it is executing. In terms of portability, this means there must be a different version of the program for each platform or the program must be able to detect the system at runtime and modify its algorithm for the particular system. Neither of these is an attractive solutions.

Another problem is that many caching systems are multilevel. A cache aware solution must be aware of every level to be effective at each level. Although this can be achieved, the resulting data structures are complex. A structure that does not require this level specific tuning, but still achieves good caching at each level, would be a much cleaner solution.

A solution to these problems is algorithms and data structures that have good caching characteristics but do not use information about the architecture of the cache. This is what Frigo, Leiserson, Prokop and Ramachandran [FLPR99, Pro99] propose. They define a ***cache oblivious*** algorithm to be one that does not know the size or layout of the cache. Algorithms can be constructed with the knowledge that a cache exists and with some basic ideas of how it operates, but not the specific details of the cache architecture. These algorithms have good, often optimal, caching characteristics. To analyze cache oblivious algorithms, Frigo et al. introduce a new caching model. While the model makes some strong sounding assumptions, Frigo et al. show that these assumptions are reasonable. Although the main focus of their paper is algorithms, they suggest that the same ideas can be applied to data structures.

There are many important data types and a correspondingly large number of implementations of them as data structures. Most of these implementations do not take caching into account. When they do it is almost exclusively in a cache aware manner; and so there are many problems that can be investigated in an oblivious fashion. As the area of cache oblivious data structures is fairly new, it is interesting to study simple abstract data types. These simple types are often used within more complex data structures. Studying the simple structures may lead to solutions to the more complex ones.

In this thesis, existing data structures are modified to improve their caching characteristics in an oblivious fashion. The work presented here, particularly in Chapter 4, is theoretical and is probably not directly applicable. Many of the results presented are asymptotically optimal. However, this notation does hide significant constants. Although these results are interesting in themselves, they also suggest more generally applicable approaches to cache oblivious data structures. A major contribution of this work is the exploration of the idea of using the recursively rotated list of Frederickson [Fre83] in the implementation of a cache oblivious linear list. Hopefully the important ideas presented here will be useful in creating new, more usable structures. This work is complementary to the approaches by others, [BDFC00], to produce entirely new cache oblivious data structures.

In Chapter 2 caching will be discussed in greater detail and the caching model will be described. In Chapter 3 a static search tree will be presented. This structure allows searches to be performed in $O(\log n)$ time and incurs $O(\log_L n)$ cache misses. This structure will also be extended to implement a heap. The heap will allow Insert, Heapifies and ExtractMaxes in $O(\log n)$ time, while incurring only $O(\log_L n)$ cache misses.

Chapter 4 describes a linked list structure which allows for a scan of a list to be performed in $\Theta(n)$ time and incurs $\Theta(\frac{n}{L})$ cache misses, where L is the length of a cache line. Insertion and deletion times are not constant in this structure, but they are in $o(n^\epsilon)$ for any constant $\epsilon > 0$. Finally Chapter 5 discusses possible areas for further study.

Notation and Conventions

In this thesis terms being defined will be written in bold italics, like ***this***. Variables from pseudocode will be indicated in non-bold italics like *this*. It is necessary to use multi-variable asymptotic notation, such as $O(f(x, y))$. This notation means if all variables but one are fixed, then the standard asymptotic equations hold for the remaining variable.

The following conventions will be used in this thesis:

- All arrays begin indexing at 0
- All logarithms are base 2 unless otherwise indicated

Chapter 2

Caching

2.1 Introduction

Caching is one attempt to reduce the waiting period imposed on a fast processor by slow memory. As discussed in Chapter 1, the high speed of modern processors and the relatively slow speed of memory access means that hundreds or even thousands of cycles could be wasted while waiting for a single memory access. The basics of caching can be found in many textbooks on modern computer architectures, such as [PH94].

The basic cache structure is shown in Figure 2.1. There are two levels of memory, main memory which is arbitrarily large[1] it but slow and the cache which is faster but has a fixed size. Both main memory and the cache are divided into blocks of fixed size. When an element in memory is accessed the cache is checked for the element's block. If the block is in cache, then the element can be read from the

[1]In theory at least. In practice it will be a large fixed size

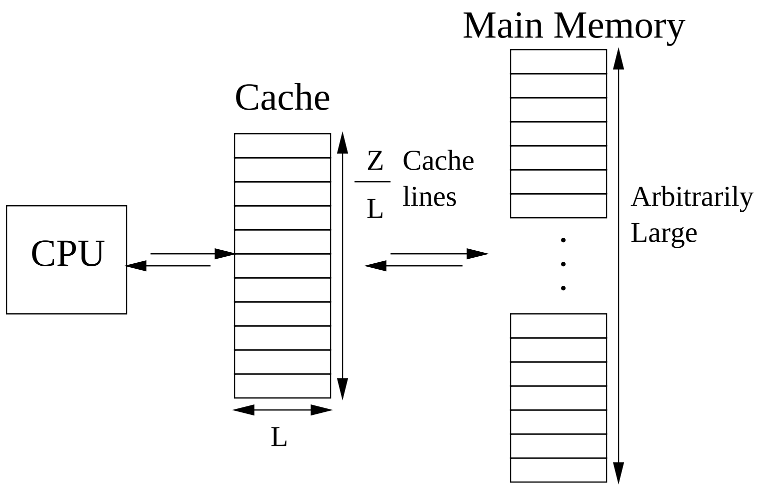

Figure 2.1: The Basic Cache Structure.

cache and an access to main memory is not required. Having an element in cache when it is accessed is called a ***cache hit***. If the block is not in cache then an access to main memory is used to copy the block from main memory into the cache. This is called a ***cache miss***.

Let Z be the size of the cache in words and L be size of a block, also in words. Define a ***cache line*** as the space in the cache which stores one block from main memory. Clearly, there are $\frac{Z}{L}$ cache lines. In practice the size of the cache is usually $\Omega(Z^2)$, and so the number of cache lines is usually $\Omega(L)$. A cache where this is true is called ***tall***. Due to the limited size of the cache, it may not be able to store all of the required data. Therefore if a cache miss occurs when the cache is full, the contents of one cache line will need to be overwritten.

In real systems the problem can be even worse. Ideally a cache would be ***fully associative***, meaning that any block from memory can be placed into any cache line. However the process of checking each cache line for a particular block can be too slow. Instead, some systems use a ***n-way set associative*** cache. In this system a block can be placed anywhere within a set of n lines. Therefore, when checking to see if a block is in cache, only n lines need be checked. The downside of this method is that cache misses can occur even if there are free cache lines.

Another key aspect of caching is the replacement technique. This is an algorithm which defines how the cache is updated when a new block must be added. The most important aspect of this is deciding which blocks to replace when there are no free lines. Clearly this is an online problem, since the algorithm cannot know what requests it will have in the future. As well, the algorithm must not take too long when deciding which block is to be replaced. These constraints lead to the development of algorithms based on simple heuristics, the most popular being Least Recently Used (LRU). The LRU algorithm makes the obvious choice for replacement, choosing the line whose contents have not be accessed for the longest time.

More advanced caching architectures will use multi-level caches. A multi-level cache has multiple caches which get smaller and faster as they get closer to the CPU. A level i cache will act as a cache for the level $i+1$ cache (level 1 being the cache closest to the CPU). Using a multi-level cache can further reduce the cost of a cache miss since the data may reside in the next level of cache.

2.2 The Ideal Cache Model

To analyze the caching characteristics of algorithms it is important to have a good theoretical model of the caching system. Frigo et al. [FLPR99, Pro99] suggest the

ideal cache model. This model is similar to a real cache as described in the previous section, with some simplifications introduced. These simplifications make the analysis much cleaner. However, each of these simplifications must be justified for the model to be valid.

The ideal cache model, like most modern computers, has a fixed word size. The total size of the cache is Z words. The length of a cache line is L words, where $L > 1$. Main memory is also divided into blocks of length L. These blocks are always transfered between memory and the cache as a single unit.

The processor can only access data stored in the cache, therefore every cache miss forces a new block to be placed into the cache. If the cache is full then some older block must be evicted. The ideal cache model uses the optimal offline replacement strategy, in which the cache line whose contents are not going to be accessed for the longest time is used to store the new block.

The ideal cache model is also assumed to have a fully associative cache. To defend this choice Frigo et al. show how to simulate such a system in main memory. Their implementation has $O(1)$ expected time accesses and uses $O(Z)$ space. Thus a fully associative system can be simulated with at worst a constant factor overhead.

Let $Q(n; Z, L)$ be the number of cache misses of a problem of size n, using a cache of size Z and cache line length of L and the optimal offline replacement algorithm. Let $Q^*(n; Z, L)$ be the number of cache misses when using LRU. Using a result from Sleator and Tarjan [ST85], Frigo et al. show that if an algorithm satisfies a simple regularity condition, $Q(n; Z, L) = O(Q(n; 2Z, L))$, then $Q^*(n; Z, L) = \Theta(Q(n; Z, L))$. This regularity condition states that doubling the cache size does not significantly effect the caching characteristics of the algorithm. Therefore the number of cache misses incurred by using the optimal offline strategy is bounded by a constant times the number used by LRU. This is important because the LRU replacement strategy, or a modification of it, is commonly used in real caching systems.

This regularity condition is important because it allows the data structure to avoid some of the worst case behavior of LRU. Consider the well known example of LRU's worst case behavior; accessing, in a cyclic pattern, $\frac{Z}{L}+1$ elements which reside in distinct memory blocks. After having accessed the first $\frac{Z}{L}$ elements, the cache will be full. To access the next element, a line must be evicted. The least recently used line will be the first one read into memory. That line will be evicted and the new one will replace it. However, the next element to be accessed will be the one in the block that was just evicted. Accessing it causes another cache miss. The block evicted for this cache miss will be the block that was accessed second, which will also be the block that is to be accessed next. This pattern will continue as long as elements are accessed in this order. Each access will cause a cache miss. Therefore, the total number of cache misses is $\Theta(n)$ for n accesses. This simple situation does not satisfy the regularity condition because doubling the size of the cache will alleviate this behavior, reducing the number of cache misses to $\frac{Z}{L}$.

Another simplification is that the ideal cache model has only two levels of memory, main memory and cache memory. However Frigo et al. show that if an algorithm is cache oblivious in a two level memory system it will also be oblivious in a multilevel system. The basic idea is that each consecutive pair of memory levels acts as a two level memory.

2.3 Simple Caching Results

Arrays will be used throughout this thesis; therefore knowing the basic caching characteristics of arrays is going to be important. First consider accessing an array in an arbitrary location and then scanning L consecutive positions (in either direction). The first access into the array causes a cache miss. Since this access is at an arbitrary location in memory the location of the element within the cache line cannot be determined. To finish the scan a second cache miss may be required.

It is useful to be able to assume that the accessed element is always in a particular position in the cache line (usually the first location). This can be assumed if $\Theta(1)$ cache misses is acceptable. In fact using two cache misses suffices. Let i be the index that is assumed and j the actual location. Then if $i < j$ there will be elements at the end of the cache line that are assumed to be in cache which are not. When one of these elements is accessed another cache miss will occur and the elements will be brought into cache. Now all the elements that were assumed to be in cache are in cache. The case when $j < i$ is similar.

Another useful observation is the minimum number of cache misses required to scan an array. An array of length n can be scanned in at most $\frac{n}{L} + 1$ cache misses. Scanning an array is an example of an operation with good caching characteristics. Every cache miss except for the first and the last brings L elements into cache. Each of these L elements is used. Therefore the total number of cache misses used is in $\Theta(\frac{n}{L})$.

Chapter 3

A Cache Oblivious Binary Search and Heap

In his Masters Thesis, Prokop [Pro99] suggests a layout for a complete binary tree that incurs $O(\log_L n)$ cache misses for a search. This layout can be implemented using a pointer based system which allows searching of the tree in $O(\log n)$ time. Concurrent with the work reported here, Bender et al. [BDFC00] implemented a similar layout as a component in a cache oblivious B-Tree. Their solution requires significant extra space. This chapter describes a layout that has the same cache complexity and running time of Prokop's layout but does not require extra storage. In addition, this structure can be created from a sorted array in linear time. Beyond the n spaces required for the data, only a constant number of indices are required for the structure. The Search procedure requires only $O(\log \log n)$ extra memory.

3.1 Layout

When studying binary search algorithms and heap structures certain types of binary trees are of particular interest. The first type, sometimes referred to as ***complete*** [CLR91] or ***perfect*** [LD91], has exactly $2^i - 1$ nodes and is laid out in perfect triangular shape (it has 2^{i-1} leaves). The second type, sometimes referred to as ***heap shape*** or (rather confusingly) ***complete*** [LD91], has all of its levels complete except possibly the last level which is full from left to a point. These two ideas will be used to build a new layout that will achieve good caching characteristics. For the purposes of this thesis the first definition of complete is the one that will be used.

To begin the description of the layout, first define a ***recursive level*** i ***complete tree*** (or a ***complete tree of recursive level*** i). This will be a complete binary tree on $2^{2^i} - 1$ nodes. A recursive level 0 complete tree is a single node. A recursive level 1 complete tree is simply a balanced tree of 3 nodes, but it is also viewed as being constructed by taking a recursive level 0 complete tree and attaching two recursive level 0 complete trees as its children. A recursive level 2 complete tree is constructed by attaching recursive level 1 complete trees as the children of the 4 leaves of a recursive level 1 complete tree, which gives a tree of 15 nodes. In general a recursive level $i + 1$ complete tree is constructed by taking a recursive level i complete tree and for each of its leaves attaching a recursive level i complete tree as the leaf's left child and another as the leaf's right child. The recursive level i trees used to construct a recursive level $i + 1$ tree will be referred to as its ***components***. Finally define the ***leaf layer*** of a complete recursive level i

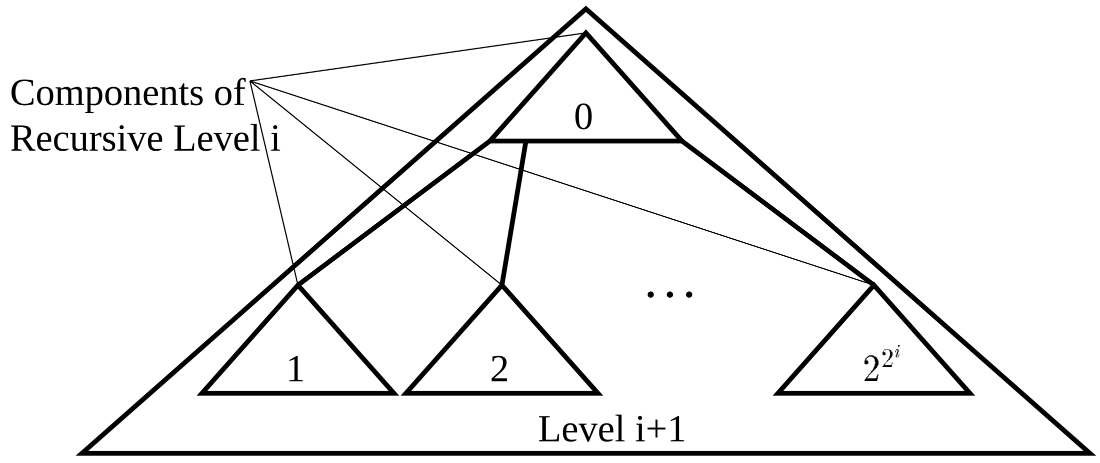

Figure 3.1: Constructing a recursive level $i{+}1$ tree from recursive level i components

tree to be all the nodes of depth 2^i. An illustration of the construction is given in Figure 3.1. The recursive form of this layout corresponds closely to the tree structure proposed by van Emde Boas [vEB75, vEBKZ77] for a priority queue.

It is easy to show by induction that a recursive level i complete tree has $2^{2^i} - 1$ nodes. For $i = 0$ we have $2^{2^0} - 1 = 1$. A recursive level $i{+}1$ complete tree consists of $2^{2^i} + 1$ recursive level i complete trees. Therefore it has $(2^{2^i} - 1) \cdot (2^{2^i} + 1) = 2^{2^{i+1}} - 1$ nodes.

A structure of arbitrary size is far more useful than a recursive level complete tree. Informally, a ***recursive level tree with*** n ***nodes, (rlt***(n)***)*** is a recursive level $\lceil \log\log(n+1) \rceil$ tree with some of the "trailing" sub-trees missing. More formally it is constructed as follows: Let $r = \lceil \log\log(n+1) \rceil$, be the recursive level of the tree and let $m = \lfloor n/(2^{2^{r-1}} - 1) \rfloor$ be the number of complete sub-trees. Take a recursive level complete $r - 1$ tree as the root sub-tree and attach $m - 1$ recursive level complete $r - 1$ subtrees contiguously from the left. Finally attach

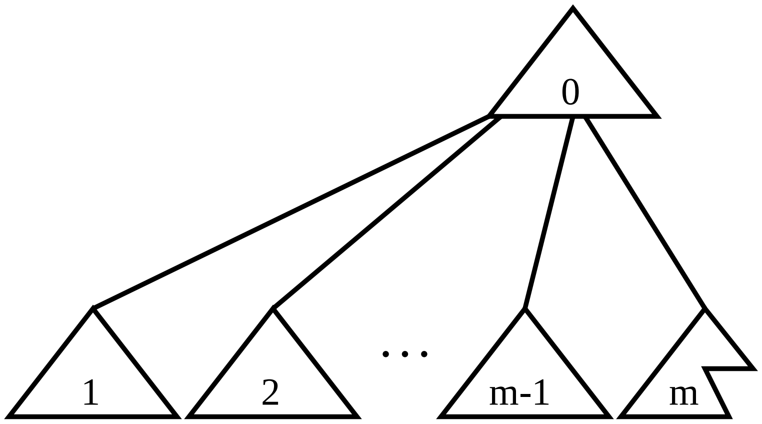

Figure 3.2: An recursive level i tree. The subtrees 0 to $m-1$ are complete recursive level $i-1$ trees and m is a rlt$(n - m \cdot 2^{2^{r-1}} - 1)$.

an rlt$(n - m \cdot 2^{2^{r-1}} - 1)$ as the next child. Figure 3.2 shows an rlt(n).

It will be useful to define an ordering for the components of an rlt(n) with recursive level r. Label the sub-tree which contains the root 0. The remaining components are labeled from 1 at the leftmost component to m at the rightmost, where $1 \leq m \leq 2^{2^{r-1}}$. For a recursive level complete tree all the components are recursive level complete trees and $m = 2^{2^{r-1}}$. For an rlt(n) there are $m = \lfloor n/(2^{2^r} - 1) \rfloor$ recursive level complete trees and possibly one rlt$(n - m \cdot 2^{2^{r-1}} - 1)$.

This tree can be represented implicitly in an array. To convert an rlt(n) with recursive level i to its array representation, recursively represent each of its components and then append them in the order defined by the labeling. A tree of recursive level 0, a single node, is represented by the value at the node. The position of a node in the array in this representation will be called its ***index***. Notice that encoding a tree of n nodes uses n indices of an array. Therefore, since the index of the first node in the array is 0, any index that is greater than or equal to the number of nodes in the tree does not represent a node in the tree.

There are two important observations to make about this structure. First, a recursive level r complete tree is a complete binary tree, therefore the height of a recursive level complete tree is the same as the height of a complete binary tree with the same number of nodes. Therefore the height of a recursive level r complete tree is 2^r. The second observation is a $\text{rlt}(n)$ has height bounded above by $2 \log n$. To see this notice that the height of a level r $\text{rlt}(n)$ is bounded above by 2^r, the height of a recursive level r complete tree. As well $n > 2^{2^{r-1}} - 1$, the number of nodes in a recursive level $r - 1$ complete tree. Therefore $h < 2 \cdot 2^{r-1} < 2 \log n$.

A cache aware method for binary search divides the array into sub-arrays of size L. Each sub-array represents a balanced subtree of the original search tree. Once a sub-array is in cache, it can be searched without any cache misses. Cache misses only occur after a search in one sub-array is completed and another needs to be pulled into cache. Searching within a single cache line allows the algorithm to move down the search tree $\log L$ steps. Therefore this arrangement requires $\lceil \log_L n \rceil$ cache misses. A simple information theoretic argument can be used to show this is optimal.

The layout presented here uses this idea, but implicitly uses recursive level r complete subtrees where $r = \lfloor \log \log(L + 1) \rfloor$ is the recursive level of the largest recursive level complete tree that fits into a single cache line. In the worst case, the cache utilization can be quite low, however the height of the subtrees is at least one half the height of the complete binary tree with L nodes. Therefore each cache miss still allows moving down the search tree $\Theta(\log L)$ elements.

The standard binary search algorithm uses an implicit tree of height $\log n$. To move from a parent to a child requires a simple constant time calculation and so the running time of the search is $\Theta(\log n)$. In the new layout, a constant time calculation to find the child of a node has not been found. However an algorithm exists to descend from any node to any leaf in its subtree in $O(\log n)$ time. Using this algorithm a binary search can be performed in the new layout in $O(\log n)$ time. The caching characteristics of the standard binary search algorithm is poor. If implemented using a sorted array every cache miss but the first returns only one useful element. Therefore the standard binary search algorithm incurs $\log n - \log L$ cache misses in a search to the bottom of the tree; i.e. in every unsuccessful search and in most successful ones.

To quickly find the children of a node, more positional information is required than just the node's location in the array. Any node in a recursive level r $\mathrm{rlt}(n)$ is contained within one of the tree's recursive level $r-1$ components. Within the recursive level $r-1$ component the node is also contained in a recursive level $r-2$ component, similarly for all recursive levels down to 0. Define the ***leaf depth*** of a node to be the largest j such that the node is in the leaf layer of a recursive level j component. Figure 3.3 shows the leaf depth of some nodes. Node **Y** is the root of a level 1 tree so the level of the largest tree it is in the leaf layer of is 0. Node **X** is the root component of a level 2 tree and the leaf layer of a level 1 tree so its leaf depth is 1. Node **Z** is in the leaf layer of a level 2 tree and the root component of a level 3 tree thus its leaf depth is 2. Then the children of this node, and any node in the leaf layer are roots of components of recursive level j.

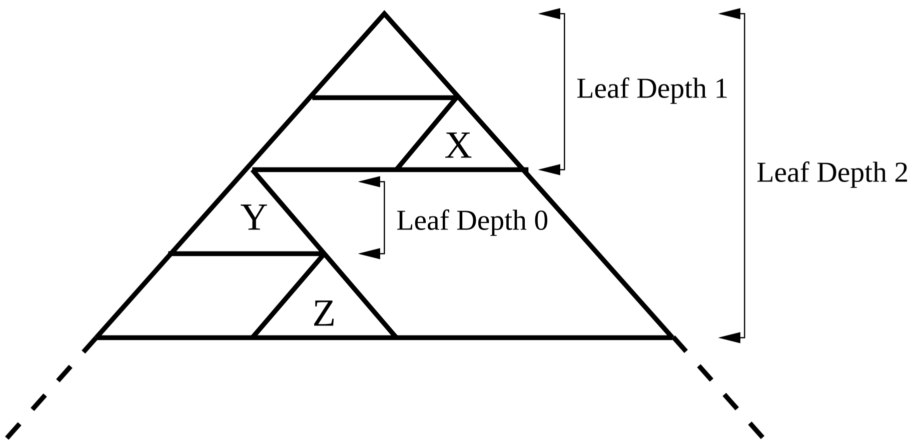

Figure 3.3: Examples of leaf depth.

Define the ***leaf indent*** of a node with leaf depth j to be the number of nodes in the leaf layer of the recursive level j subtree to the left of the node. Then twice the leaf indent of a node is the number of recursive level j subtrees to the left of the children of the node, Figure 3.4 illustrates this. Each of these subtrees contains $2^{2^j} - 1$ nodes. Knowing the index I of the root of the recursive level j tree in which the node resides and the leaf indent of the node, the indices of its children can be calculated using the following equations,

$$\begin{aligned} C_L &= I + (2^{2^j} - 1) \cdot (2 \cdot indent + 1) \\ C_R &= I + (2^{2^j} - 1) \cdot (2 \cdot indent + 2) \end{aligned}$$

Therefore to calculate the index of the children of a node one needs to know:

- j, the leaf depth of the node.
- I, the index of the root of the tree.

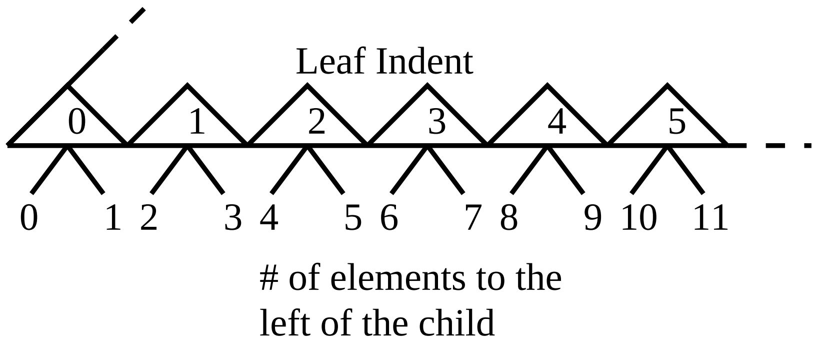

Figure 3.4: Examples of leaf indent.

- *indent*, the leaf indent of the node.

To find j one needs to know in which components the node resides. To do this use a table of size $\lceil \log \log n \rceil + 1$ called *location*. This table holds the labels of the components which contain the node ($location[i]$ is the label of the component of recursive level $i-1$ within the component of recursive level i). Indices to the roots of each of the components are also needed. These will be stored in another table, of the same size, called *offset*. An example of these tables is given in Figure 3.5. The following procedure computes these tables. It assumes that the *location* and *offset* tables are global variables. It also assumes a global *level* variable exists which stores the recursive level of the tree.

```
FindLocation($index$)
    $lev \leftarrow level$
    $offset[lev] \leftarrow 0$
    $location[0] \leftarrow 0$
    while ($lev > 0$)
        $size \leftarrow 2^{2^{lev-1}} - 1$
        $location[lev] \leftarrow \lfloor \frac{index - offset[lev]}{size} \rfloor$
        $offset[lev-1] \leftarrow offset[lev] + location[lev] \cdot size$
```

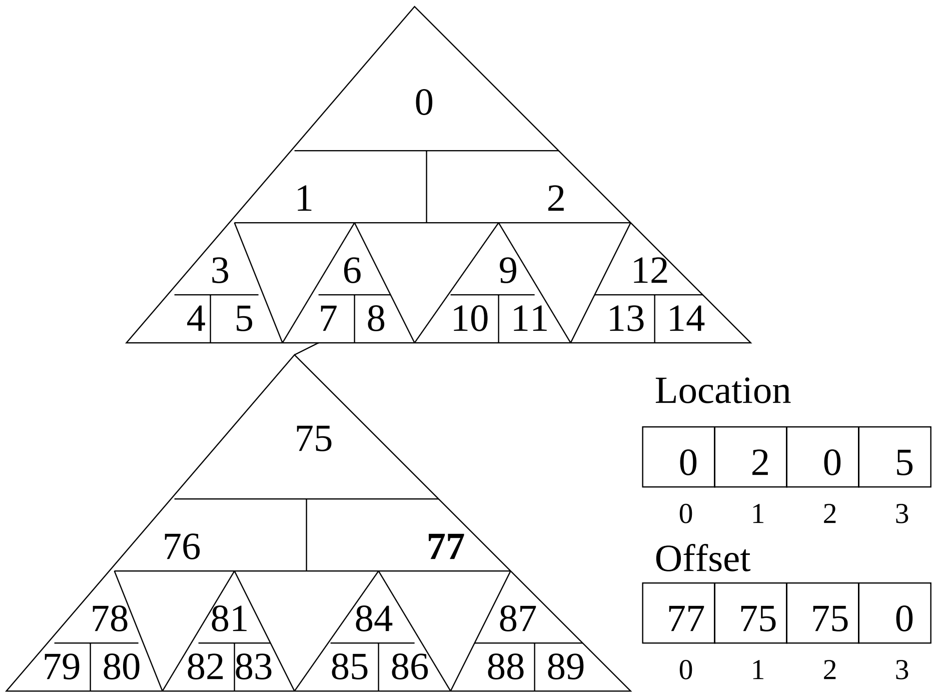

Figure 3.5: An example of the location and offset tables for the element with index 77

```
        $lev \leftarrow lev{-}1$
```

The work done before the while loop is constant. One iteration of the while loop is also takes constant time, therefore the running time of FindLocation is $\Theta(level) = \Theta(\log\log n)$, the number of iterations of the while loop. The algorithm makes $\Theta(level)$ accesses into various arrays. These accesses have decreasing indices starting from *level* going down to 0. The cache complexity of stepping through these arrays is $\Theta(\frac{level}{L}) = \Theta(\frac{\log\log n}{L})$.

Given the *location* and *offset* tables, the leaf depth and leaf indent of the node, v, must be calculated. The leaf depth of v can be found by looking through the table for the smallest i such that $location[i] = 0$. This is the first subtree for which v is not in its leaf layer. Therefore the leaf depth is $i - 1$. The leaf indent can be calculated by using the following observation: in a component j of a recursive level k tree, the number of nodes in the leaf layer of the tree to the left of the leftmost node of j is the number of leaves in a recursive level $k - 1$ component times the number of components to the left of j. By the labeling, the number of such subtrees is $j - 1$. To calculate the indent of a node sum the indents of the leftmost nodes of each of the recursive level i components, for $1 \leq i \leq depth$. Notice that v is the leftmost node of the subtree of recursive level 0. The following procedure calculates both the indent and the depth. As before assume that *location*, *offset*, *indent* and *depth* are global variables.

```
CalcIndent()
    $indent \leftarrow 0$
    $i \leftarrow 1$
    while ($location[i] > 0$)
        $indent \leftarrow indent + 2^{2^{i-1}-1} \cdot (location[i] - 1)$
    $i \leftarrow i+1$
  $depth \leftarrow i-1$
```

The running time of this function is $\Theta(depth)$. The cache complexity of this function is $\Theta(\frac{depth}{L})$. Notice that if the first L elements of *location* are already in the cache and *depth* is less that L, then this function incurs no cache misses.

The following two procedures return the left and right child of the current location.

```
CalcLeftChild()
  return $offset[depth]+(2^{2^{depth}}-1)\cdot(2\cdot indent+1)$

CalcRightChild()
  return $offset[depth]+(2^{2^{depth}}-1)\cdot(2\cdot indent+2)$
```

The FindLocation procedure is too slow to use for every node as the search moves down the tree. Instead of re-evaluating each time, the location and offset tables can be modified to reflect the changes when moving to a child. Before moving down the node is on the leaf layer of each of the *depth* subtrees. After the move down, the new node is in the label 0 component of all the *depth* subtrees. This means for $0 \leq i \leq depth$, $location[i]=0$. Since the child node is the root of all of these subtrees, the correct offset value is the child node's current value. The child node is no longer within the $depth+1$ subtree labeled 0, however we can easily calculate which of its children we are in since we know the leaf indent of the node. The new label for $depth+1$ is $2\cdot indent+1$ if moving left or $2\cdot indent+2$ if moving right. In the following pseudocode implementation of MoveDown, the variable *left* is a boolean variable which represents whether *child* is the left child.

```
MoveDown($child$, $left$)
    for $i$ in 0 to $depth$
        $location[i] \leftarrow 0$
        $offset[i] \leftarrow child$

    if ($left$)
        $location[depth{+}1] \leftarrow 2{\cdot}indent{+}1$
    else
        $location[depth{+}1] \leftarrow 2{\cdot}indent{+}2$
```

This procedure also has running time $\Theta(depth)$ and cache complexity $\Theta(\frac{depth}{L})$. As before if $depth \leq L$ and the first L elements of *location* already in cache then this function incurs no cache misses.

3.2 "Binary" Search

This Section will describe one use for the cache oblivious tree layout, a cache oblivious search. The Search procedure will have optimal running time and caching characteristics. A linear time method to convert a sorted array into this layout is also presented.

The Search procedure is basically identical to the standard binary search procedure, except using the more complex child finding procedures.

```
Search($value$)
    FindLocation(0)
    CalcIndent()
    if ($value = array[0]$)
        return true
    $next \leftarrow 0$
    while ($next < length$)
        if ($value > array[next]$)
            $next \leftarrow$ CalcLeftChild()
            $left$ ← true
        else if ($value$ < $array[next]$)
            $next$ ← CalcRightChild()
            $left$ ← false
        else
            return true
        MoveDown($next$,$left$)
        CalcIndent()
    return false
```

Theorem 1 *Search takes $O(\log n)$ time.*

Proof: To begin the proof the contribution from the while loop will be calculated. Let h be the height of the tree. Consider the location array as holding zero or non-zero values; then it encodes the height of the location it represents in binary. The larger the index into the table, the higher the order of the bit (we will not consider the recursive level 0 element as it is always 0 and never changes). A move down increments this "binary counter" (replace the initial block of consecutive non-zeros with zeros and change the next zero to a non-zero). So moving from the root, (all zeros) to a leaf (encoding of h) is work equivalent to incrementing a binary counter h times. The work required to increment a binary counter h times is the total number of bits flipped. For half of the increments 1 bit is flipped, for a quarter 2 bits are flipped, for $1/2^i$ of the increments i bits are flipped.

Let $b = \lceil \log h \rceil$, the number of bits in the counter. Then the work done incrementing a binary counter h times is bounded by

$$\begin{aligned} \sum_{i=0}^{b} \frac{h}{2^i} &= h \sum_{i=0}^{b} \frac{1}{2^i} \\ &\leq h \sum_{i=0}^{\infty} \frac{1}{2^i} \end{aligned}$$

$$\begin{aligned} &= h\frac{1}{1-\frac{1}{2}} \\ &= 2 \cdot h \end{aligned}$$

Therefore the running time of the while loop is $O(h)$. Since the height of the tree is at most $2 \log n$, the running time is $O(\log n)$.

The call to FindLocation and the first call of CalcIndent contribute $\Theta(\log \log n)$ to the running time and rest of the function is constant time. Hence using this layout we can perform searches in $O(\log n)$ time. □

Theorem 2 *If $L \geq 5$, Search incurs $O(\log_L n)$ cache misses.*

Proof: To show $O(\log_L n)$ cache complexity, a caching scheme will be given which uses $O(\log_L n)$ cache misses for a search. Since an optimal offline caching strategy is assumed in our model, the algorithm will incur at most this many cache misses. The number of cache lines required is: 1 for searching the tree and 2 for each of the tables. Therefore a total of 5 cache lines are required.

There are two sources of cache misses in this operation, the array which stores the data and the *location* and *offset* tables. The contributions from each of these will be considered separately. First consider those cache misses caused by reading from the data array.

A $\text{rlt}(n)$ of recursive level j can be viewed as constructed from recursive level k complete trees, $0 \leq k < j$, and a small number of $\text{rlt}(m)$ trees, $1 \leq m < 2^{2^k} - 1$. To view a tree this way recursively consider the components of the tree until the components are of recursive level k. For $r = \lfloor \log \log(L+1) \rfloor$, r is the recursive level of the largest recursive level complete tree that can fit into a single cache line. View a tree using the given layout as consisting of subtrees of this size.

A recursive level r tree can be brought into cache using 2 cache misses. If $n < L$ then the 2 cache misses required to load $\mathrm{rlt}(n)$ into cache suffices for the entire search. Otherwise, consider the search path. It is of length at most $2\log n$. The height of a recursive level r complete tree is 2^r, which is at least $\frac{1}{2}\log L$. Therefore the search path is covered by at most $2\log n/2^r$ complete recursive level r trees. The total number of cache misses required to follow this path is twice the number of trees, $2 \cdot 2\log n/\frac{1}{2}\log L = 8\log_L n$.

To see that this can be done in a single cache line, notice that once a cache line has been searched it does not need to be considered again. Therefore there is no reason to keep the elements in cache. They can be overwritten by the incoming elements without causing cache misses later.

Now consider the more subtle issue of the *location* and *offset* tables. Looking over the pseudocode one notices that the two tables are accessed in slightly different ways. The *location* table is scanned from 0 to $depth + 1$ twice. The *offset* table is accessed once at $depth + 1$ and then scanned from 0 to $depth$. An upper bound on how many access are required for both tables is two scans from 0 to $depth + 1$. Using this upper bound the same analysis can be used for both tables.

The caching scheme used for the tables is to place first L table elements into the cache. If accesses to more than the first L elements are required, then, as often as needed, load the required values into the cache line. After they have been used, preload the first L table elements back into cache. (Alternatively the next access will be for the first element in the table so first L elements will be cached then.) Since the first L elements may not be within a single block, two cache lines are required to guarantee that these elements are in cache.

Now from the equations from above the total number of table accesses is $\sum_{j=0}^{b} \frac{h}{2^j}$, where $b = \lceil \log h \rceil$ as before. Assume $L < b$, that is, the tables are larger than a single cache line (otherwise it would only require a single cache miss to load the entire table into the cache). So the total number of table accesses which occur within the first L elements of the table is $\sum_{i=0}^{L} \frac{h}{2^i}$ Therefore the total number of table accesses which occur outside the first L elements is

$$\begin{aligned}
\sum_{i=0}^{b} \frac{h}{2^i} - \sum_{j=0}^{L} \frac{h}{2^j} &= h\left(\sum_{i=0}^{b} \frac{1}{2^i} - \sum_{j=0}^{L} \frac{1}{2^j}\right) \\
&= h\left(\frac{(\frac{1}{2})^{b+1} - 1}{\frac{1}{2} - 1} - \frac{(\frac{1}{2})^{L+1} - 1}{\frac{1}{2} - 1}\right) \\
&= h\left(\left(\frac{1}{2}\right)^{L} - 2 - \left(\frac{1}{2}\right)^{b} + 2\right) \\
&= h\left(\frac{1}{2}\right)^{L} - h\left(\frac{1}{2}\right)^{b} \\
&= \frac{\log n}{2^L} - \frac{\log n}{2^{\log\log n}} \\
&= \frac{\log n}{2^L} - 1 \\
&= O\left(\frac{\log n}{2^L}\right)
\end{aligned}$$

So even if every table access that occurs outside the first L elements causes a cache miss the number of misses caused by the tables is much smaller than is used in searching the tree. Therefore the total cache complexity of a search in this layout is $O(\log_L n)$. □

Given a sorted array a search tree using this layout can be built in $O(n)$ time. No claims about the caching characteristics of this procedure will be made. To begin the description, a method for converting a sorted array with $2^{2^i} - 1$ (for any integer $i \geq 0$), elements into a recursive level complete tree will be presented. The basic idea of this method is to place the $\frac{n}{2}$ leaves of the recursive level complete tree in their correct locations while dividing the remaining elements into sorted arrays. The algorithm will then recursively insert these arrays into the new layout.

The goal of this algorithm is to convert the sorted array representation of a binary search tree to the new layout. Doing this will not change the position of elements in the implicit tree, only the indices of the elements in the representation are modified. Therefore the leaves in the sorted array representation are also leaves in the new representation. In the sorted array, the leaves are in the even indices of the array. To add the leaves to the new layout, their correct position in the new layout must be determined.

From the recursive description of the layout, a recursive description for finding the indices of the leaves can be found. Let i be an integer greater than zero. The leaves of a recursive level i complete tree are the leaves of its recursive level $i-1$ components with label greater than 0. Since the tree is recursive level complete the component with label 0 has no leaves. When the recursion reaches a recursive level 0 complete tree, that node is a leaf.

Using this recursive definition, the new tree layout can be stepped through from back to front, in a manner that touches all leaves without touching the internal nodes. Recursively inspect each of the components in order from the highest labeled component to the component labeled 1 and then skip the component labeled 0. This can also be done without recursion. To do this, start at the end of the array and move towards the front; if the next element is in a component labeled 0, skip the entire component.

This iterative method can be used to find the correct position for the leaves in the new representation. However, the internal nodes also need to be added. If instead of skipping, as described above, these elements are stored then they can be inserted recursively. To store these internal nodes a two dimensional array will be used. An array will store the elements of a label 0 component that has not been added to the new layout. Therefore there will be arrays of size $2^{2^i} - 1$ for $0 \leq i < level$ where *level* is the recursive level of the tree.

A label 0 component will be written out after the label 1 component is added. To determine when a label 0, recursive level i component should be added, a check is performed after each recursive level i component is added. If the array storing the label 0 recursive level i component is full, then it should be added. Notice that after adding a label 0 recursive level i component, the addition of a recursive level $i+1$ component has been completed. Therefore this check is performed within a loop which increments the recursive level as label 0 components are added.

For this method to work, the algorithm must be able to determine where to put elements from the input array. If the index into the input array is even, then it is a leaf and should be added directly into the new layout. Otherwise store the node in the array corresponding to the recursive level i label 0 component, where i is the recursive level of the last tree added to the new layout.

The following pseudocode will illustrate the conversion procedure. The variable *compZero* is the two dimensional array containing *level* arrays. The array *compZero*$[i]$ is of length $2^{2^i}-1$. This array will be used to store elements for the recursive level i component labeled 0. The variable *compZeroIndex* is an array such that *compZeroIndex*$[i]$ is the current insertion point in *compZero*$[i]$. Elements will be added to the *compZero* arrays from the last element to the first. Since the algorithm decrements through the sorted input array, this will guarantee the elements are added in sorted order.

```
LevelCompleteBuild($data$, $level$)
    LevelCompeteBuildHelper($data$, $level$, 0, $2^{2^{level}}-1$)


LevelCompeteBuildHelper($data$, $level$, $low$, $high$)
    $arrayIndex \leftarrow high$
    $dataIndex \leftarrow 2^{2^{level}}-2$
    Set $compZeroIndex[i] \leftarrow 2^{2^i}-2$ for $0 \le i < level$
    while ($arrayIndex \ge low$)
        if $dataIndex$ is even
            $array[arrayIndex] \leftarrow data[dataIndex]$
            $dataIndex \leftarrow dataIndex$-1
            $arrayIndex \leftarrow arrayIndex$-1
            $last \leftarrow 0$
        else
            while ($compZeroIndex[last] = -1$)
                $end \leftarrow arrayIndex$
                $arrayIndex \leftarrow arrayIndex - 2^{2^{last}}-1$
                LevelCompleteBuildHelper($compZero[last]$, $last$, $arrayIndex$, $end$)
                $arrayIndex \leftarrow arrayIndex - 1$
                $compZeroIndex[last] \leftarrow 2^{2^{last}}-2$
                $last \leftarrow last+1$
            $compZero[last][compZeroIndex[last]] \leftarrow data[dataIndex]$
            $dataIndex \leftarrow dataIndex$-1
        $compZeroIndex[last] \leftarrow compZeroIndex[last]-1$
```

Theorem 3 *A sorted array with $2^{2^i}-1$ elements can be converted into a recursive level i complete tree in $O(n)$ time.*

Proof: The given algorithm adds all of the leaves of the input tree to the new layout. The internal nodes are added recursively. The amount of work done, not considering the recursive calls, is proportional to the size of the input array. The total number of recursive calls is hard to calculate, however the total number of elements recursed upon is less than $\frac{n}{2}$. Let R be the set of sub-arrays that are recursed upon. Then the following is a recurrence for the running time of this algorithm.

$$\begin{aligned} L(n) &= n + \sum_{r \in R} L(\text{length of } r) \\ L(1) &= 1 \end{aligned}$$

A proof by induction will show $L(n) < 2n$. This is clearly true in the base case. Assume it is true for values less than n. Then recurrence becomes,

$$\begin{aligned} L(n) &= n + \sum_{r \in R} L(\text{length of } r) \\ &\leq n + 2\sum_{r \in R}(\text{length of } r) \\ &\leq n + 2(\text{the number of elements recursed upon}) \end{aligned}$$

$$\begin{aligned} &\le n + 2\frac{n}{2} \\ &\le 2n \end{aligned}$$

Therefore the algorithm is $O(n)$. □

To build a tree with n elements, for any n, use LevelCompleteBuildHelper as a subroutine. Given a tree in the sorted array representation, its recursive level, l, can be easily computed. A rlt(n) of level l is made of one or more complete trees of recursive level $l-1$ and one rlt of level $l-1$. The recursive level complete subtrees can be laid out using LevelCompleteBuildHelper and the incomplete recursive level $l-1$ subtree can be laid out recursively. The elements in the components with labels greater than 0 are stored contiguously in the input array, the elements for the label 0 components are not. These elements are interlaced between the elements for the components with label greater than 0. When there are fewer than $2^{2^{l-1}}$ such components, there are elements at the end of the input array which are not allocated to a component. These elements are elements of the recursive level 0 component.

```
Build($data$, $n$)
    $level \leftarrow \lceil \log\log n \rceil$
    BuildHelper($data$, $level$, $0$, $n$)
```

```
BuildHelper($data$, $level$, $offset$, $n$)
    $s \leftarrow 2^{2^{level-1}} - 1$
    $numOfCompleteComp \leftarrow \lfloor \frac{n}{s} \rfloor$
    $sizeOfIncompleteComp \leftarrow n \bmod s$
    for $i = 0$ to $numOfCompleteComp - 2$
        $low \leftarrow (s+1)i$
        $high \leftarrow (s+1)i + s - 1$
        Copy elements $data[low...high]$ into a temporary array $tmp$
        (This could also be achieved using pointers into the $data$ array)
        LevelCompleteBuildHelper($tmp$, $level-1$, $s(i+1)$, $s(i+2)-1$)
        $compZero[i] \leftarrow data[high+1]$
    $low \leftarrow (s+1)(numOfCompleteComp - 1)$
    $high \leftarrow low + sizeOfIncompleteComp - 1$
    Copy elements $data[low...high]$ into a temporary array $tmp$
    BuildHelper($tmp$, $level-1$, $offset + low$, $sizeOfIncompleteComp$)
    Copy the elements $data[low + sizeOfIncompleteComp...n-1]$ to the end of $compZero$
    LevelCompleteBuildHelper($compZero$, $level-1$, 0, $s-1$)
```

Theorem 4 *A sorted array of length n can be converted into the new layout in $\Theta(n)$ time.*

Proof: Let $s = 2^{2^{l-1}} - 1$, the size of a recursive level $l-1$ complete tree. Let $L(n)$ be the running time of LevelCompleteBuildHelper. Then a recurrence for the running time of this algorithm is as follows,

$$T(n) = \left\lfloor \frac{n}{s} \right\rfloor L(s) + T(n \bmod s) + n$$

The running time for LevelCompleteBuildHelper is linear so this recursion simplifies to

$$T(n) = T(n \bmod s) + O(n)$$

An upper bound on $n \bmod s$ must be determined. To make the number of elements recursed upon as large as possible (in terms of n), the number of recursive level complete sub-trees should be as small as possible. Therefore if there is one recursive level complete sub-tree and one incomplete one, the number of elements that is recursed upon is at most $\frac{n}{2}$. Given this, the recursion simplifies even further and the result is a simple application of the Master Theorem [CLR91]. $\square$

This section has described a cache oblivious layout which encodes a static binary search tree into an array. This layout allows searches in $O(\log n)$ time and incurs $O(\log_L n)$ cache misses, thus it is time and cache optimal.

3.3 Heaps

Another application of our layout is a heap-like implementation of a priority queue with good cache characteristics. The major difference between binary search and the heap is the necessity of moving up the tree as well as down. A Parent function will be introduced that calculates the parent of a node, as well as a MoveUp function that will modify the *location* and *offset* tables when the algorithms need to go to a parent. These functions will then be used to implement the standard priority queue functions.

To calculate the index of the parent of a node a procedure similar to finding the children will be used. As before the discussion will start with some definitions. Define ***root depth*** to be the recursive level of the largest component that has the node as its root. Figure 3.6 illustrates this. The node **X** is in the leaf layer of a level 1 tree therefore the largest component that has **X** as a root is 0. Node **Y** is the root of a level 1 tree but not a level 2. Therefore **Y**'s root depth is 1. Similarly node **Z** is the root of a level 2 tree, but not level 3 and so its root depths is 2.

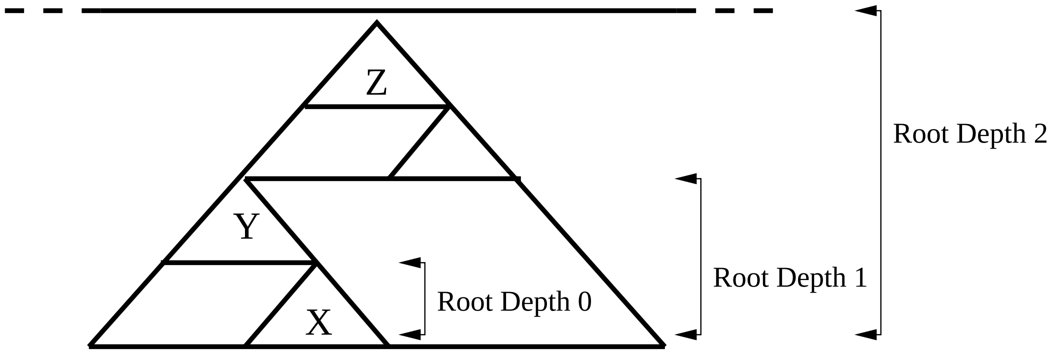

Figure 3.6: Examples of root depth and root indent

Define ***root indent*** to be the label of the node's level *root depth* component (the one the node is a root of) in the recursive level *root depth* $+ 1$ tree. Finding the root depth is similar to finding the leaf depth; simply scan through the *location* table until the first non-zero entry is found. The first non-zero entry is the first tree for which the node is not in the root component. Since the node was in the root component for all the lower level trees it was the node for each of them. The root indent is the label of the component within the *root depth* $+ 1$ tree. Calculating the root indent is simply a matter of taking the difference in the index of the root of the *root depth* $+ 1$ tree and the node (the number of nodes between them) and dividing by the number of nodes in a recursive level *root depth* complete tree. This counts the number of level *depth* complete sub-trees that are between the root of the level *depth* $+ 1$ tree and the node. The CalcParentIndent function will be used to find these values.

```
CalcParentIndent()
    $i \leftarrow 1$
    while ($location[i] = 0$)
        $i = i+1$
    $depth = i-1$
    $indent = \lfloor \frac{(index - offset[depth+1])}{2^{2^{depth}}-1} \rfloor$
```

The running time of this function is $\Theta(depth)$.

The root indent of a node indicates the subtree in which it resides. The parent of this node is the $\lfloor \frac{indent-1}{2} \rfloor^{th}$ node from the left in the label 0 component. The label 0 component (of recursive level *depth*) is composed of recursive level $depth-1$ components. The number of leaves in the leaf layer of one of these components can be calculated using their depth. By dividing the parent indent by the number of leaves, the recursive level $depth-1$ component that the parent is in can be found. By taking the parent indent modulo the number of leaves in the leaf layer of a $depth-1$ component we can find the indent into this component. The offset to the root of these subtrees becomes the index of the parent node when the recursive level of the subtree is 0. The following function calculates the index of the parent.

```
Parent()
    $total \leftarrow offset[depth+1]$
    $b \leftarrow \lfloor \frac{indent-1}{2} \rfloor$
    $lev \leftarrow depth-1$
    while ($lev \geq 0$)
        $size \leftarrow 2^{2^{lev}-1}$
        $total \leftarrow total + \frac{b}{size+1} \cdot (2size-1)$
        $b \leftarrow b \bmod size$
    return total
```

Once again the running time of this function is $\Theta(depth)$.

Calling FindLocation every time the algorithm move to a parent is too slow. Therefore a MoveUp function is used which updates the *location* and *offset* tables to the parent of a node in $\Theta(depth)$ time. This function is nearly identical to the FindLocation function, the difference being it only modifies the first *depth* elements. Notice the MoveDown function is a similar modification, except since the first *depth* elements must be 0, their values need not be calculated.

```
MoveUp($index$)
    $location[depth+1] \leftarrow 0$
    $offset[depth] \leftarrow offset[depth+1]$
    $lev \leftarrow depth$
    while($lev > 0$)
        $size \leftarrow 2^{2^{lev-1}} - 1$
        $location[lev] = \frac{index - offset[lev]}{size}$
        $offset[lev\text{-}1] = offset[lev] + location[lev] \cdot size$
        $lev \leftarrow lev\text{-}1$
```

It is clear that the running time of this function is also $\Theta(depth)$.

Inserts are performed as normal with the addition of the new navigation functions. The element is added to the end of the array and then moved up the tree until the heap property is achieved.

```
Insert($val$)
    if ($numOfElements = 2^{2^{level}} - 1$)
        $level \leftarrow level+1$
    $i \leftarrow numOfElements$
    $numOfElements \leftarrow numOfElements+1$
    FindLocation($i$)
    CalcParentIndent()
    $par \leftarrow$ Parent()
    while ($i > 0$ AND $array[par] < val$)
        $array[i] \leftarrow array[par]$
        $i \leftarrow par$
        MoveUp($par$)
        CalcParentIndent()
    $par \leftarrow$ Parent()
$array[i] \leftarrow val$
```

To analyze the running time and cache complexity of this function apply the ideas used in the proofs for Search's complexity. First notice that the path traveled from the leaf to the root for an Insert is the path that a Search to that leaf would have taken in reverse order. Also notice that the leaf depth of a node is the same as the root depth of its child. This means that calling MoveDown at some node takes similar time and memory accesses to calling MoveUp from its child to get to the node.

Theorem 5 *Insert has running time $O(\log n)$.*

Proof: Outside of the while loop, Insert calls FindLocation and CalcIndent and does a constant amount of other work. So the cost outside the while loop is $\Theta(\log \log n)$. Using the observations above, the sum of the leaf depths moving down a path is the same as the sum of the root depths moving up a path. Therefore the while loop does asymptotically the same amount of work as the while loop from Search moving down that path. Therefore the while loop takes $O(\log n)$ and so the Insert function takes $O(\log n)$ time. □

Theorem 6 *Insert has cache complexity $O(\log_L n)$.*

Proof: The analysis of the cache complexity of Insert is also similar to that of Search. There are two source of cache misses; the accesses into the data array and the accesses into the *offset* and *location* tables. For Search the number of table accesses was bounded by two scans through an array from 0 to *depth+1* per step.

In this case three are required. Notice that each of CalcParentIndent, Parent and MoveUp access the *offset* array. However this is only a constant multiple difference and so does not affect the asymptotic number of cache misses. Since the leaf depth for a transition from a node to its child is the same as the the root depth when moving from child to parent, the total number of accesses into the tables for a path from the root to a leaf is the same as the path from the leaf to the root. Therefore the result obtained in the proof for Search's cache complexity applies and so the total number of cache misses caused by table accesses is $O(\frac{\log n}{2^L})$.

Now consider the number of cache misses contributed by the data array. Let $r = \lfloor \log\log(L+1) \rfloor$. As in the analysis for Search a complete recursive level r tree can be brought into cache with 2 cache misses. The height of the level r tree is at most $\frac{1}{2}\log L$. As well, the length of the path from the new leaf to the root is at most $2\log n$. Thus the number of cache misses required to move the element up the heap is at most $8\log_L n$. □

The Heapify function is important to maintain the heap property after extracting the maximum element. Heapify assumes that FindLocation has been called before its first invocation.

```
Heapify($index$)
    calcChildIndent()
    $left \leftarrow$ LeftChild($index$)
    $right \leftarrow$ RightChild($index$)
    $which \leftarrow$ true
    if ($left < numOfElements$ AND $array[left] > array[index]$)
        $larger \leftarrow left$
    else
    $larger \leftarrow index$
    if ($right < numOfElements$ AND $array[right] > array[larger]$)
```

```
        $larger \leftarrow right$
        $which \leftarrow$ false
    if ($larger \neq index$)
        Swap the values of $array[index]$ and $array[larger]$
        MoveDown($larger$, $which$)
        Heapify($larger$)
```

Theorem 7 *Heapify has $O(\log n)$ running time.*

Proof: Proving the running time for Heapify is very similar to the proofs from previous theorems. Descend the tree calling by CalcChildIndent and MoveDown at each step does $O(depth)$ work per step. Using the calculations from above the total running time of Heapify is $O(\log n)$. □

Theorem 8 *Heapify has cache complexity $O(\log_L n)$.*

Proof: As before let $r = \lfloor \log\log(L+1) \rfloor$. Notice that, unlike previous algorithms, Heapify must access a node's left and right child to decide how to proceed. This means that it will access array positions and then decide to not proceed to them. Previous proofs assumed that complete level r trees were brought into cache with 2 cache misses. In a complete tree the only nodes which do not have left and right children in cache are the leaves. Therefore only when moving between the complete level r sub-trees are extra cache misses required and then only one additional miss per sub-tree. Instead of charging 2 cache misses per sub-tree, charge 3. The total number of cache misses required to walk the path is $3 \cdot 2 \log n / \frac{1}{2} \log L = 12 \log_L n$. The tables are used as before and so contribute a lower order term. □

Since we know Heapify's caching and running time characteristics it is easy to find ExtractMax's characteristics.

```
ExtractMax()
    if ($numOfElements$ < 1)
        Error("Heap Underflow")
    $max \leftarrow array[0]$
    $array[0] \leftarrow array[numOfElements]$
    $numOfElements \leftarrow numOfElements$-1
    if ($numOfElements = 2^{2^{level-1}} - 1$ AND $numOfElements > 0$)
        $level \leftarrow level$-1
    FindLocation(0)
    Heapify(0)
    return max
```

Theorem 9 *ExtractMax has running time $O(\log n)$ and cache complexity $O(\log_L n)$.*

Proof: Before the call to FindLocation, the function uses a constant amount of time and cache misses. FindLocation takes $\Theta(\log \log n)$ time and $\Theta(\frac{\log \log n}{L})$ cache misses. Thus the running time and cache complexity of this function is dominated by Heapify. Therefore ExtractMax has $O(\log n)$ running time and has cache complexity of $O(\log_L n)$. □

To build a heap in $O(n)$ time a similar method as was presented for the search tree will be used. As before no claims about the caching characteristics of this algorithms are made. To begin the discussion consider an input array containing $2^{2^i} - 1$ elements. A heap storing these elements will be a level i complete tree. Such an array can be converted into a heap stored as a level i complete tree in $O(n)$ time. First run the standard make heap algorithm to reorganize the array such that it is stored as a heap. By performing an in order walk of this tree and writing the elements into an array, they are converted into the layout used for the input to the LevelCompleteBuild algorithm from Section 3.2. Using LevelCompleteBuild this array will be rearranged into the new layout. All of the steps required to convert the array into the new layout take $O(n)$ time, so the entire operation takes $O(n)$.

```
LevelCompleteHeapBuild($data$, $n$, $low$, $high$)
    MakeHeap($data$, $n$)
    $tmp$ = InOrderWalk($data$, $n$)
    LevelCompleteBuildHelper($tmp$, $n$, $low$, $high$)
```

This procedure will be used as a subroutine for the general building procedure. Most input arrays will not be of length $2^{2^i}-1$. As with building the search tree, an input array will be divided into a set of sub-trees which are of size $2^{2^{j-1}}-1$, where j is the level of the rlt(n), and one sub-tree of smaller size that is laid out recursively.

To make an unsorted array into a heap, the values stored in a parent must be larger than those stored in its children. Therefore if the top $2^{2^{j-1}}-1$ elements are selected and assigned to be the root sub-tree, then any heaps made from the remaining nodes can be placed as the children of any of the leaves of this sub-tree. By using the linear selection algorithm of Blum, Floyd, Pratt, Rivest and Tarjan [BFP+73, CLR91] the k^{th} element can be found in linear time. By partitioning the input array using this element the k largest elements can be split from the $n-k$ smallest. The remaining nodes will be divided into groups of size $2^{2^{j-1}}-1$ and possibly one of smaller size. The root sub-tree and the groups of size $2^{2^{j-1}}-1$ can be laid out using LevelCompleteHeapBuild. The one group of smaller size can be laid out recursively.

```
HeapBuild($data$, $n$)
    HeapBuildHelper($data$, $n$, 0)
HeapBuildHelper($data$, $n$, $low$)
    $level \leftarrow \lfloor \log\log(n+1)$
    $SubTreeSize \leftarrow 2^{2^{j-1}} - 1$
    $k \leftarrow$ Select($data$, $n$, $SubTreeSize$)
    ($topK$, $rest$) $\leftarrow$ Partition($data$, $n$, $k$, $SubTreeSize$)
    LevelCompleteHeapBuild($topK$, $SubTreeSize$, $low$)
    $i \leftarrow SubTreeSize$
    while ($i + SubTreeSize < n$)
        $tmp \leftarrow rest[i,\ldots,i+ SubTreeSize]$
        LevelCompleteHeapBuild($tmp$, $SubTreeSize$, $low + i$)
        $i \leftarrow i + SubTreeSize$
    $tmp \leftarrow rest[i,\ldots,n]$
    HeapBuildHelper($tmp$, $n - i$, $low + i$)
```

The running time analysis is identical to the one used in the proof of Theorem 4. Therefore this algorithm is also $\Theta(n)$. Thus a heap can be built using the new layout in linear time from an array of n numbers.

Chapter 4

Cache Oblivious List

4.1 Introduction to Rotated Lists

The linear list is one of the simplest abstract data types. Standard implementations allocate nodes such that moves forward and backward, as well as insertions and deletions take constant time. However, these methods lead to the list having poor caching characteristics. A scan of the entire list can take n cache misses, one miss for each node in the list. A data structure that can be used to represent a linear list and also has good caching characteristics is desirable. The data structure presented here will allow a list to scanned in $\Theta(n)$ time and incur the optimal number of cache misses, $\Theta(\frac{n}{L})$, without knowing L. It also allows insertions and deletions in time $O(n^{\frac{1}{\sqrt{\log n}}}\sqrt{\log n})$. The extra space used by this new structure, over and above the data itself, can be made as small as ϵn for any constant $\epsilon > 0$. This is in contrast to the work of Bender, Demaine and Farach-Colton on B-trees [BDFC00] which requires considerable extra storage. Most notably, they make use of buffer nodes which substantially increase the storage requirements. This structure requires no such nodes. To achieve these results, the recursively rotated list, a structure proposed by Frederickson [Fre83], will be used. This structure is a generalization of the rotated lists of Munro and Suwanda [MS80].

Rotated lists were proposed by Munro and Suwanda to solve the dictionary problem with no structural information (pointers). They store their data in nearly sorted order and allow fast structural operations. Like Frederickson's generalization, the structure presented here exploits these properties to store the nodes of a list in nearly contiguous order while allowing updates in $O(n^{\frac{2}{\sqrt{2\log n}}}\sqrt{\log n})$ time. As a linear list is implemented instead of a dictionary, there are some differences between the operations of the two data types. The most important difference is that inserting into a linear list has both the value to be inserted and the location. A dictionary is given only the value and must itself decide where to put the element. As the original application stores elements ordered by value the description of recursively rotated lists will as well, however once the new structure is described in Section 4.3 this will not longer be the case.

A rotated list is a sorted array whose which has had a cyclic shift (shifted to the right with the elements shifted off the end moved to the spaces opened at the beginning) applied to its elements. Instead of the first element being at index 0 and the last at index $n-1$, the first element is at index s, the $n-s$th element is in index $n-1$, the $n-s+1$th element is in index 0 and the nth element is in index $s-1$. For a rotated list it is useful to understand how far the array has been rotated. To do this, define the ***rotation element*** of a rotated list to be the

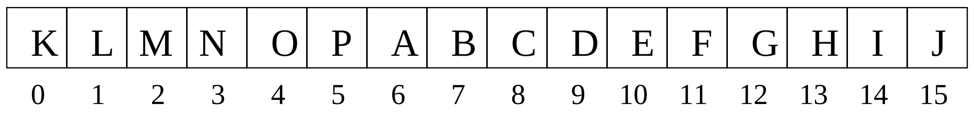

Figure 4.1: A rotated list of 16 elements.

physical location or index of the logically last or largest element in the array. This element indicates how far the array has been rotated. In the example of a rotated list given in Figure 4.1 the rotation element occurs at position 5, as the list has been shifted 6 locations.

There are two important operations that can be performed on a rotated list, the easy exchange and the hard exchange. An ***easy exchange*** is inserting an element that is smaller than any other element in the rotated list and deleting the largest element. A ***hard exchange*** is inserting an element whose value is larger than the smallest value in the list and deleting the largest element.

An easy exchange can be implemented in constant time by replacing the largest element in the rotated list with the new smallest one. Figure 4.2 is an example of an easy exchange[1]. To perform a hard exchange, find the location for the element to be inserted and replace the value stored in that location with the new one and the insert the old value into the next position. Continue shifting values back through the list until the last position is reached. Since the largest value is being deleted its value can be overwritten with the new one. Figure 4.3 illustrates this procedure. Clearly this operation can take time linear in the length of the rotated list. During either of these operations there will an element that is not stored within the array,

[1]In examples, letters will be used as data items. They are in alphabetical order

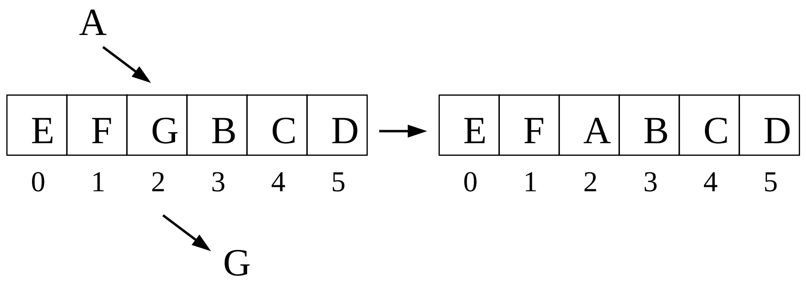

Figure 4.2: An easy exchange: inserting A, deleting G.

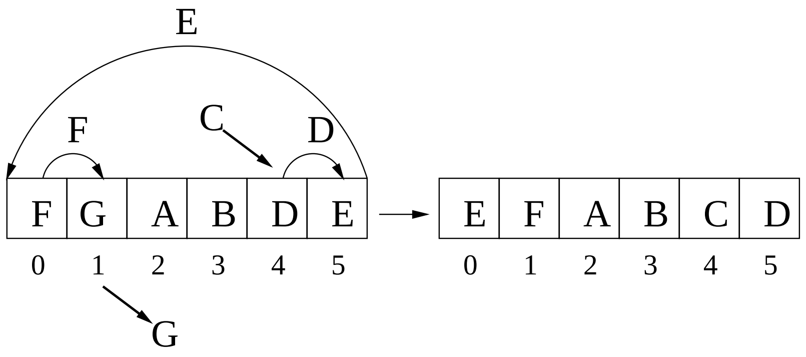

Figure 4.3: A hard exchange: inserting C, deleting G.

either the new element being inserted or the old element being deleted or in the case of a hard exchange one of the elements that is being shifted. Call this element the ***shift element***.

Inserting into a rotated list means adding an element in an arbitrary location while maintaining the order of the elements. Given that the rotated list is represented within an array, the insertion will need to move elements so that a space at the end of the array is used. Therefore performing an insert in a rotated list is similar to performing a hard exchange. The difference is that instead of moving elements towards the rotation element they are moved towards the end of the ar-

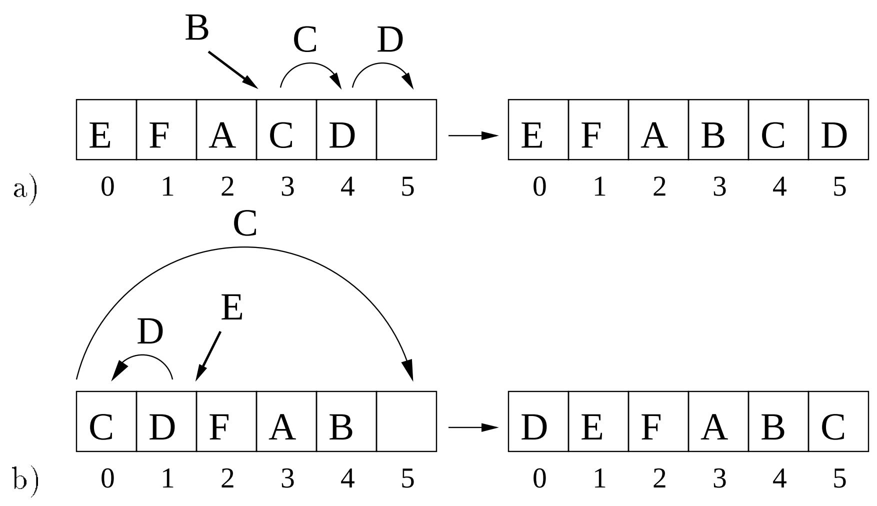

Figure 4.4: Insertion into a rotated list. Figure 4.4a is an insertion to the right of the rotation element. Figure 4.4b is an insertion to the left of the rotation element.

ray. At the end of this procedure the shift element will be the element that should be placed at the end of the array. Once this element is found it can be added to the end of the array. Performing insertions in this way maintains the rotated list structure.

It maybe desirable for the position of the rotation element not to change during an insert operation. Therefore if the insertion occurs before the rotation element, instead of shifting towards the end of the array one can shift towards the beginning. Once the front of the array is reached the shift element can be placed at the end of the array and still maintain the rotated list structure. Examples of the two insertion directions are given in Figure 4.4. This operation is also linear in the length of the rotated list.

A deletion is completely analogous to an insertion. The element to be deleted is removed. To fill the gap shift elements either forwards or backwards depending on the location of the deletion relative to the rotation element. If elements are being shifted from the back, when the end of the array is reached the algorithm can terminate. If elements are being shifted from the front, when the first element is reached the element from the end of the array is removed and used to fill the first location.

4.2 Multi-Level Rotated List Structures

4.2.1 Two Level Structures

Munro and Suwanda [MS80] use rotated lists to make a new structure, the list of rotated lists. A list of rotated lists is a structure which stores a set of sorted elements using multiple rotated lists. The structure is represented in an array. The smallest element in the set is stored at the beginning of the array in a rotated list of length 1. The next two smallest elements from the set form a rotated list of length 2 stored in the array after the rotated list of length 1, then next 3 are stored in a rotated list of length 3. In general, the $(\sum_{i=1}^{k-1} i)^{th}$ to $\left((\sum_{i=1}^{k} i) - 1\right)^{st}$ smallest elements are stored in a rotated list of length k stored in the array starting at position $\sum_{i=1}^{k-1} i$. The last rotated list may not be full; in this case it is represented as a rotated list containing the remaining elements. Figure 4.5 is an example of a list of rotated lists.

To insert an element perform a hard exchange on the rotated list in which the insert occurs. If the value of the element falls between two lists then the initial hard exchange is not required. The shift element is smaller than any element in the

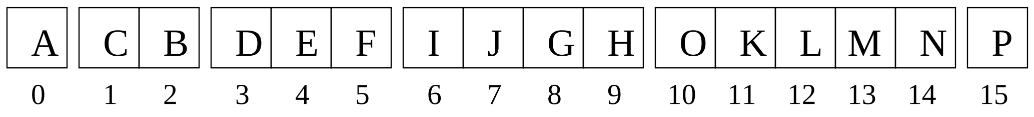

Figure 4.5: A list of rotated lists containing 16 elements.

following rotated list so an easy exchange can be performed. The largest element from the rotated list becomes the shift element after the easy exchange. Continue performing these easy exchanges until the last rotated list is reached. Perform an insert into the last rotated list. If this final rotated list is full, then create a new rotated list of one element and add it to the end of the list[2].

As before, deletes are similar to inserts. Remove the element to be deleted from the rotated list. Then perform a hard-exchange-like operation which moves the free space to the end of the rotated list. To fill this position take the smallest element from the next list. By the properties of the list of rotated lists this element is larger than any element in the previous list therefore placing it in the last position maintains the properties of a list of rotated list. To fill the now empty position in this rotated list take the smallest element from the next rotated list. Since the gap was created by removing the smallest element a new element larger than any in the list can be placed in that position, similar to an easy exchange. Continue performing these operations until the last rotated list is reached. The smallest element from this list was used to fill the gap in the previous list so its position can be deleted.

Frederickson [Fre83] generalized the list of rotated lists by noticing that the array

[2]Memory management will be discussed in Section 4.5.1

could be divided in a more arbitrary manner. Given a function $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$ and n the number of elements to be represented, define r to be the smallest integer such that $n \leq \sum_{i=1}^{r} f(i)$. Divide the array, of size n, into r sub-arrays, $S(1), S(2) \ldots S(r)$. Each of the sub-arrays, $S(j)$ with $1 \leq j < r$ contains the $f(j)$ elements from the array in positions $\sum_{i=1}^{j-1} f(i)$ to $(\sum_{i=1}^{j} f(i)) - 1$. The remaining elements from position $\sum_{i=1}^{r-1} f(i)$ to n are stored in $S(r)$. The elements in the sub-arrays can be stored using a secondary structure.

Using this method the list of rotated lists can be viewed as using the function $f(i) = i$. Therefore $n \leq \frac{r(r+1)}{2}$ and so r is approximately $\sqrt{2n}$. The running time of an insert is the time to perform the insert in the last rotated list plus the time for the easy exchanges. There are at most r rotated lists so the number of the easy exchange is $O(\sqrt{n})$ and the size of the last rotated list is $O(\sqrt{n})$, so the running time of an insert in this structure is $O(\sqrt{n})$.

4.2.2 Recursively Rotated Lists

Frederickson further generalized these ideas into recursively rotated lists [Fre83]. A recursively rotated list of recursive level h is a rotated list that has been divided into sub-arrays, $A(0), A(1), \ldots, A(B(h)-1)$, where $B(h)$ is the branching factor of level h. Each of these sub-arrays are then represented as recursively rotated lists of recursive level $h-1$. A recursively rotated list of recursive level 0 is a rotated list of c elements (where c is an arbitrary parameter). The total number of elements in a recursively rotated list of recursive level h is $c\prod_{i=1}^{h} B(i)$, (c can be viewed as $B(0)$). The elements in the $B(h)$ sub-arrays of a recursively rotated list satisfy two properties. Let p be the index of the sub-array such that the rotation element of the rotated list appears in $A(p)$.

1. $x < y$, for all $x \in A(i)$ and all $y \in A((i+1) \bmod B(h))$ for $0 \leq i < B(h)$ and $i \neq p, i \neq (p+1) \bmod B(h)$.

2. $z \in A(p)$ implies $z < y$ for all $y \in A((p+1) \bmod B(h))$ or $x < z$ for all $x \in A((p+1) \bmod B(h))$.

The first condition states that for two adjacent rotated lists, neither of which contains the rotation element for this level, all the elements in the first list are smaller than all the elements in the second. This is a similar constraint to the one imposed in a list of rotated list. The second constraint states that any element from the sub-array containing the rotation element is either larger than every element in the next list or smaller than every element from the next list. This is reasonable since the sub-array containing the rotation element will contain some of the largest elements in the list and in general some of the smallest.

A sub-array which does not contain the rotation element from the level above is simply a sorted array. Therefore the initial position of its rotation element is the last position in the sub-array. If the sub-array does contain the rotation element of the level above, then the initial position of its rotation element is the location of the rotation element of the level above. In either case the sub-array is a valid recursively rotated list.

These two possibilities for the initial position of the rotation element raise an important issue. There are two concepts of the order of elements in a recursively rotated list. One idea is the order in which the elements are stored in the entire structure. For a standard recursively rotated list this is sorted order. The second idea is the order the elements appear in the array which stores the recursively rotated list. When a recursively rotated list does not inherit the rotation element from the level above these two ideas are the same. However, when the recursively rotated list does inherit the rotation element from above these ideas are different.

To distinguish these two concepts define ***list first*** and ***list last*** to be the first and last elements in a recursively rotated list that are first and last according to the order of the entire structure, (notice that the rotation element is the list last element). As well, define ***array first*** and ***array last*** to be the elements of a level i recursively rotated list that would be the first and last in the array if the level i structure were simply an array. One useful observation to make is that an inherited rotation element is a array last element from a level above. Therefore every rotation element is an array last element from a list of level greater than or equal to its level.

Since this is a dynamic structure it is possible that a level h recursively rotated list does not contain $B(h)$ sub-arrays. In this general case, a level h recursively rotated list contains d sub-arrays where $2 \leq d \leq B(h)$. The sub-arrays $A(0)$ to $A(d-2)$ are complete level $h-1$ recursively rotated lists (having $B(h-1)$ sub-arrays). The sub-array $A(d-1)$ need not be complete.

An example of a recursively rotated list

It is difficult to visualize the multiple levels of rotations that a recursively rotated list uses. This example gives a complete explanation of the rotated nature of a recursively rotated list. Figure 4.6 is the recursively rotated list that will be used

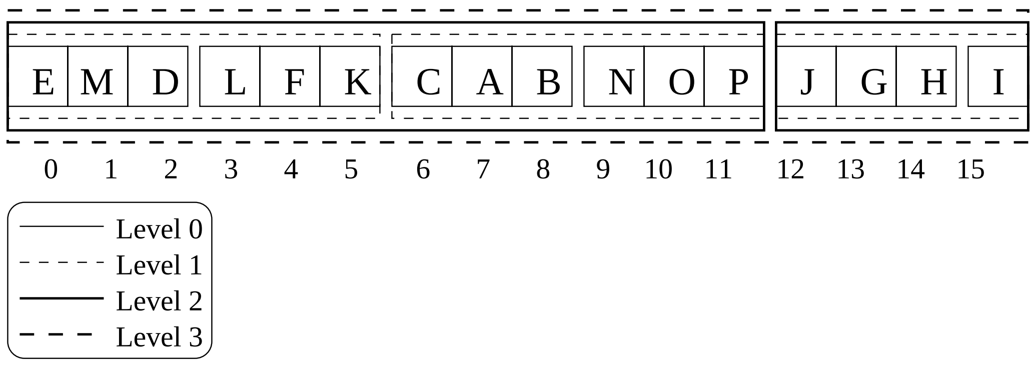

Figure 4.6: A level 3 rotated list containing 16 elements

as an example. It contains the elements **A** to **P**. This example will be constructed through a set of rotations at each level.

First, if the entire structure were a single rotated list it would be the rotated list shown in Figure 4.7. The elements have been shifted by 6 positions, moving the elements **G,H,I,J** to the end of the list. This rotated list of 16 elements will be divided into sub-arrays of length 12. Since there are only 16 elements, the array is divided into two sub-arrays, one of length 12 and one of length 4. The four elements in the second sub-array are determined by this first rotation. Since the unrotated version of the list in Figure 4.7 is simply a sorted array the array first and list first are the same as are the array last and list last.

Figure 4.8 adds the level 2 rotations (shown in the figure as a level 1 recursively rotated list). The array has been split into two sub-arrays. The first sub-array contains the rotation element, **P**, from the level above, therefore **P** is the rotation element of this rotated list. This also means that the array last is different from the list last. Since this list has been rotated by 3 positions the array last element

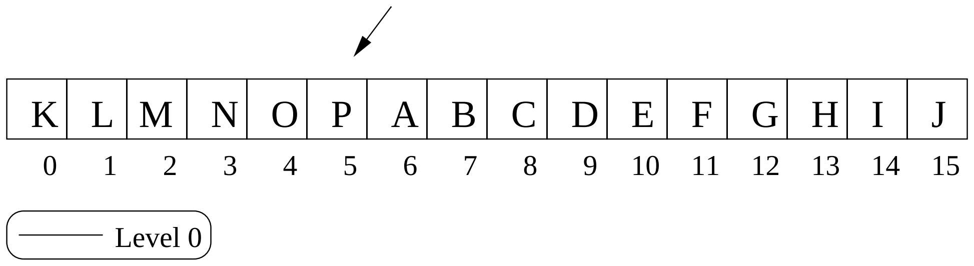

Figure 4.7: The level 3 rotation of Figure 4.6 shown as a level 0 rotated list.

is located in position 2. This diagram shows what array first and last elements indicate. The elements **K** and **F** are the array first and last elements for the first rotated list. Although they are in alphabetical order, there are elements between that are stored in a different rotated list. Therefore these elements represent a possible discontinuity that is not obvious from the order of the elements in the array.

The second sub-array does not contain the rotation element from the level above so the initial position of its rotation element is the last position in the array. Therefore the array first and list first are the same, as are the array last and list last. In this example, this sub-array is not rotated from its initial position, though it can be in general.

The next level of recursion divides the array into sub-arrays of length 6. In this case we get 3 new sub-arrays, two of length 6 and one of length 4. This is shown in Figure 4.9. The first rotated list does not contain the rotation element from the level above so it is simply a sorted array. This sub-array is rotated by 1 position, as indicated by the arrow. The next sub-array does contain the rotation element from the level above, therefore the initial position of its rotation element is the position

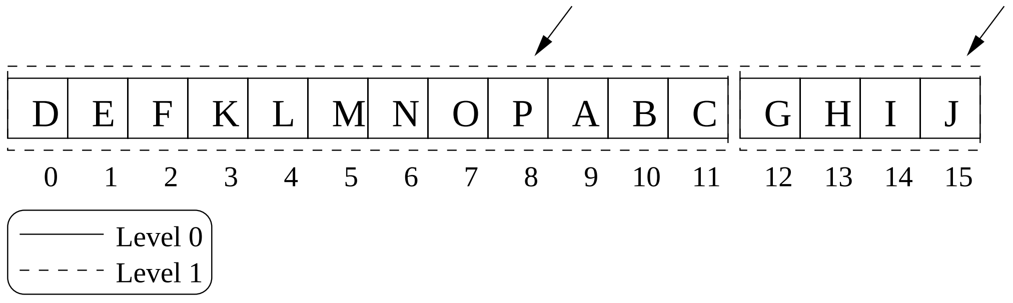

Figure 4.8: The second level of rotations. Notice the rotation element for the first rotated list is the rotation element from the level above.

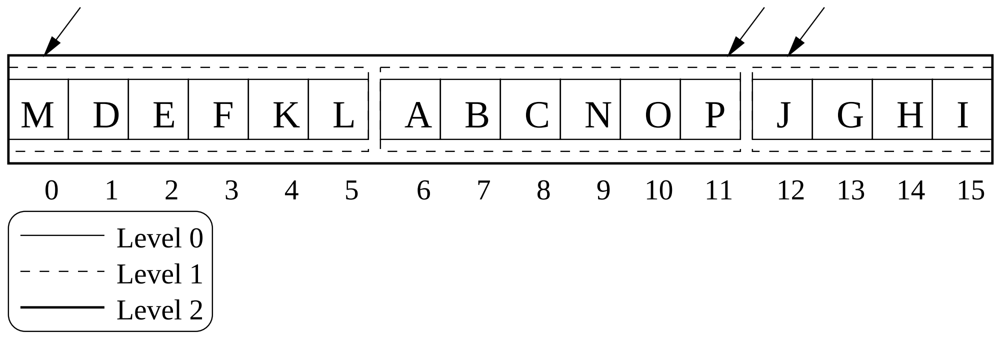

Figure 4.9: The third level of rotation.

of the rotation element from the level above. This sub-array has been rotated by 3 elements. The last sub-array also contains the rotation element from the level above (which happens to be in the last position) and is rotated by 1 position.

The final level of rotation, show in Figure 4.10, divides the array into sub-arrays of length 3. As before, the sub-arrays which do not include the rotation element from the level above are simply sorted arrays with the initial position of their rotation element in their last position. The other arrays have the initial position of their rotated element in the position from the level above.

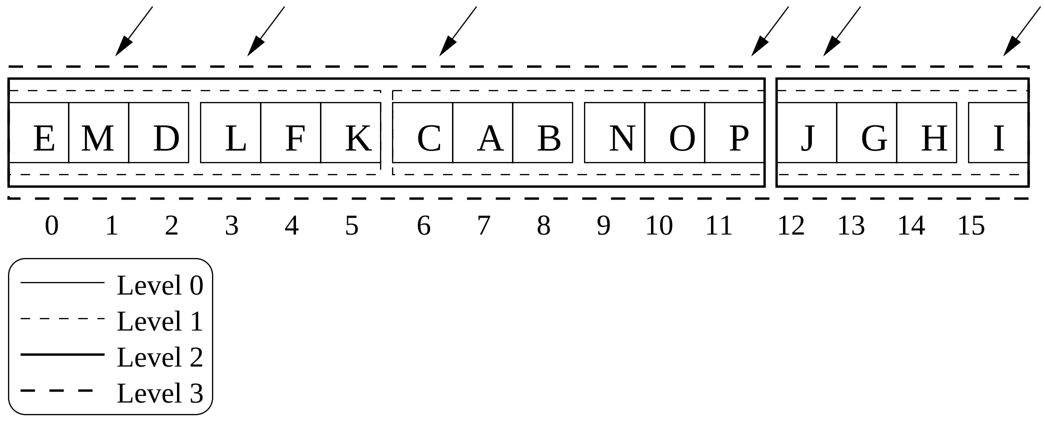

Figure 4.10: The level 3 recursively rotated list from above, notice the positions of the rotation elements.

4.2.3 Operations on Recursively Rotated Lists

An easy exchange on a recursively rotated list can be performed as before, by simply replacing the largest element with the new element. To see that this maintains the properties outlined in Section 4.2.2, notice that after an easy exchange the second to largest element is now the largest. Usually this element will be within the same rotated list, however it is possible that it is not. In the case when it is within the same rotated list, the list contains some number of the smallest elements and some of the largest elements. Therefore property 2 still holds since an element in this list is either one of the smallest, thus smaller than any other element in the rest of the list or one of the largest, thus larger than any other element in the rest of the list. Property 1 holds since none of other sub-arrays were modified. If the easy exchange does move the rotation element from one sub-array to another, the old sub-array contains the smallest elements in the list and the new one contains the largest. This means the sub-array after the array containing the rotation element contains the smallest elements in the list, thus property 2 holds. Since the sub-array containing the smallest elements no longer has the rotation element property 1 must be checked. However, this is obvious given that this sub-array contains the smallest elements in the list.

A general recursive hard exchange can be implemented as follows: First find the sub-array in which the element should reside, perform a recursive hard exchange in this list. Then perform easy exchanges in each sub-array until the sub-array containing the rotation element is reached. Perform a recursive hard exchange in this sub-array. Some cases are not covered by this general statement. If the element to be added falls between two sub-arrays then the initial hard exchange is not required since the element does not need to be inserted into an array. If the correct sub-array to add the new element into is the sub-array containing the rotation element, then only a single recursive hard exchange is required. Due to the recursive nature of this structure this sub-array is a recursively rotated list so performing a hard exchange in this sub-array does have the correct effect. The element returned from this hard exchange is the largest element in the sub-array which is also the largest element in the entire list.

```
RecHardExchange($array$, $level$, $element$)
    if ($level = 0$)
        return HardExchange($array$, $element$)
    Set $i$ to be the index of the sub-array ($S(i)$) in which the element should reside
            or the following sub-array if the element falls between two sub-arrays.
    if (the element does not fall between two sub-arrays
            AND $S(i)$ does not contain the rotation element)
        $element \leftarrow$ RecHardExchange($S(i)$, $level - 1$, $element$)
        $i \leftarrow (i+1) \bmod B(level)$
    while ($S(i)$ does not contain the rotation element)
```

```
        $element \leftarrow$ RecEasyExchange($S(i)$, $level - 1$, $element$)
        $i \leftarrow (i + 1) \bmod B(level)$
    return RecHardExchange($S(i)$, $level - 1$, $element$)
```

An example of a hard exchange in a recursively rotated list is given in Section 4.3.2[3].

A hard exchange does not violate the properties of a recursively rotated list. First notice that much of the work done by a hard exchange uses easy exchanges. Easy exchanges have already been shown to perserve the recursively rotated list properties. The rest of the work is done by performing hard exchanges in level 0 rotated lists. A hard exchange in a rotated list does not change the relative order of the elements. The position of the largest element is unchanged after a hard exchange, although the value is not. The other elements are handled similarly. Thus the properties hold since the relative order of elements is unchanged after a hard exchange in a level 0 rotated list.

Performing an insert in a recursively rotated list is the natural extension of an insert into a list of rotated lists. The main idea is to change the element and position of insertion from the original to the end of the rotated list so the element added to the end of the list maintains the properties of a recursively rotated list. The first step is to check if there is space for the new element. If there is no free space it must be allocated[4]. The actual insertion is then performed.

An insertion can be thought of as a hard exchange in the recursively rotated list containing the insertion point, followed by easy exchanges until the recursively

[3]The example includes modifications for the new structure

[4]The details of this are presented in Section 4.5.1

rotated list containing the free spaces is reached. A recursive insert is then performed in this recursively rotated list. During this sequence passing over the first or last element of the entire list is a problem. An easy exchange cannot be performed within this list as the element to be inserted is not necessarily smaller than any element within the list. An operation similar to a hard exchange could be made to work, however this is undesirable. It is faster to simply find the direction for operations which avoids this position.

Unfortunately this is not as simple as it sounds. Although this operation is simple in general, it breaks down in some cases which must be handled individually. The general idea of this operation is to determine the relative positions of the insertion point and the rotation element. If the insertion point is after the rotation point, then the insertion can proceed to the end of the array without encountering the rotation element. If the insertion point is before the rotation point then the insertion must proceed towards the beginning of the array to avoid the rotation element.

To determine the direction first check the values of the insertion point and rotation element. If the insertion point is the same as the rotation element, then move towards the front of the array. If these positions are not equal then view the level h recursively rotated list as being $1 < b \leq B(i)$ level $h-1$ recursively rotated lists where the last sub-array, A_0, may not be full. If the sub-array which contains the insertion point, I_0, is different from the sub-array that contains the rotation element, R_0, then this is the easy case. If the position of R_0 is less that the position of I_0 then the operations proceed towards the end of the array. If I_0 is before R_0

proceed towards the beginning of the array. If I_0 is the same sub-array as A_0 then simply perform an insert in the level $h-1$ array.

The difficult case is when I_0 and R_0 are the same. Consider the recursively rotated list I_0. To determine the correct direction to move the relative positions of the rotation element and the insertion point must be determined. If the insertion point is after the rotation element and before, or equal to the array last element, then move towards the end of the list. Otherwise move towards the front. Let j be the level of the recursively rotated list. Let the sub-array containing the insertion point be I, the sub-array containing the rotation point be R, and sub-array containing the list last element (for the level j recursively rotated list) be L.

If the order of these three sub-arrays is R, I, L, in the rotated sense, then the insertion point is clearly after the rotation point and before the array last element. If the order is R, L, I then the insertion point is after the array last element, but before the rotation point. Now the cases in which some of the sub-arrays are the same must be considered.

If I and R are the same but L is not, then recursively apply this procedure on this sub-array. The position of the insertion point relative to the rotation point within this sub-array will imply its position in the array. If I and L are the same but R is not, then determining the direction to move is equivalent to determining if the insertion point is before or after the array last element in this sub-array. A recursive call in this sub-array can be used to solve this case. If the insertion point is before the array last element then move towards the array last element, thus the end of the array. Otherwise move toward the array first element.

If R and L are the same, but I is not then finding the order of the three positions is equivalent to finding the order of the rotation point and the array last element within the sub-array. Therefore recursively apply this within the sub-array to determine the correct order of the positions. The final case is I, R and L all the same sub-array. This case can also be solved recursively. However, in this case the array last element from the current array is passed into the sub-array to be used, instead of its own array last value.

If the recursion reaches a level 0 sub-array then the relative order of the positions can be determined by noticing that this case is similar to having sub-arrays of length 1. Since the insertion and rotation elements are known to be different, the only case to worry about is if the insertion element and the array last element are the same, or the rotation element and the array last element. In the first case, the rotation point must be before the insertion element, so the operation can proceed towards the end of the array. In the second case, the insertion element is clearly before the rotation element and so the operation should proceed towards the front of the array.

Once the direction to move has been determined a procedure similar to a recursive hard exchange is performed in that direction. First a recursive hard exchange operation is performed in the sub-list of level $h-1$ in which the insert occurs. This operation adds the new element to the list and removes either the largest element or the smallest depending on the direction. Easy exchanges are used to shift elements until an element whose position is between the last recursively rotated list in the array and the first. The algorithm checks to see if the last sub-array is full. If it is then instead of recursively inserting, a new recursively rotated list of the same level as the sub-array is created. The algorithm will add an element to this new recursively rotated list.

If the last sub-list is not full then a recursive insert is performed in the last sub-list. If the algorithm reaches level 0 then the rotated list obtained is incomplete. Therefore the element can be inserted in this rotated list.

The following pseudocode illustrates the recursively rotated list Insert procedure.

```
RecInsert($array$, $level$, $element$)
    if ($level = 0$)
        Insert($array$, $element$)
        return
    Set $i$ to be the index of the sub-array ($S(i)$) in which the element should reside
            or the following sub-array if the element falls between two sub-arrays.
    Determine the direction to perform the operations
    if (proceeding towards the end of the array)
        if (the element does not fall between two sub-arrays)
            $element \leftarrow$ RecHardExchange($S(i)$, $level - 1$, $element$)
            $i \leftarrow (i + 1) \bmod B(level)$
        while ($S(i)$ is not the first sub-array)
            $element \leftarrow$ RecEasyExchange($S(i)$, $level - 1$, $element$)
            $i \leftarrow (i + 1) \bmod B(level)$
    else
        Similar to moving towards the end of the array, except with backward
        hard and easy exchanges. Check for the last sub-array instead of the
        first
    if (the last sub-array is not full)
        RecInsert(last sub-array, $level - 1$, $element$)
    else
        Create a new recursively rotated list of level $level$ containing only $element$
    return
```

To see that an insert does not violate the properties of a recursively rotated list, first consider the movement of elements within the rotated list. To obtain an element whose position is between two rotated lists a hard exchange is used. Easy exchanges then shift elements until an element whose correct position is between the last rotated list and the first if found. This element could be inserted into either list if there is space. However, knowing that the first list is fullonly the last needs to be checked. If it is full then a new list is created. The new list becomes the last list in the array. Therefore the value of it's elements should be between those in the recursively rotated list before it and the recursively rotated list after it. The element that was used to create the new list has this property. In the case that the insertion reaches a level 0 rotated list, then there is space for the element in the level 0 list. An insertion will add the element into this list. The level 0 rotated list is the last one in the array. Therefore adding the element to this list places at the end of the last recursively rotated list. This maintains the properties of a recursively rotated list.

Deletions can be performed using a similar procedure to insert. First determine which direction to proceed. The desired element is removed, creating a gap in a recursively rotated list. A hard-exchange-like operation is used to shift the gap to the end of the recursively rotated list. Easy-exchange-like operations can shift the gap until it is in a recursively rotated list that is adjacent to the last recursively rotated list. The last element from the last recursively rotated list is fill the gap and then it is recursively deleted.

4.3 New Structure

4.3.1 Modifications

The new data structure uses recursively rotated lists, but changes some aspects so that it represents a linear list and not a sorted array. This makes the structure explicit rather than implied by the key values. The nodes of the list are standard doubly linked list nodes. They store a data item, a next pointer and previous pointer (a method will be presented later which will reduce the required space). The element reached by following a next pointer corresponds to the next larger element in Fredrickson's dictionary. All of the list nodes are stored in an array. Within this array the nodes are stored in a recursively rotated list in list order. Since the data cannot be used to determine relative order between nodes, pointers will be stored and used to maintain the order. Each recursively rotated list will have a pointer to its array first, array last, list first and list last element. Since these modifications do not effect the order elements are stored in the recursively rotated list, the properties listed in Section 4.2.2 must still be maintained.

The layout of the list within the array is as follows. A list with recursive level 0 has a constant number of elements, c, stored as a rotated list in a contiguous block of memory. Pointers to the array first and last and the list first and last elements in the rotated list are stored in a secondary structure. A level h recursively rotated list consists of $B(h)$ level $h-1$ recursively rotated lists stored contiguously in the array. Pointers to the array first and last and list first and last elements of this list are also stored.

The pointers can be stored in four two dimensional arrays. One dimension indicates the level of the recursively rotated lists that the pointers index. The other dimension determines which recursively rotated list of that level it points into.

4.3.2 Easy Exchanges and Hard Exchanges

An easy exchange in the new structure is more complex than in a regular rotated list. As with a standard recursively rotated lists replace the last element in the rotated list with the new one; however, the pointers in the new structure must also be updated. The pointers for the list can be updated as in a standard list. To update the array first and array last pointers notice that the element that was the array first element is now the second and the element that was the second to last element is now the array last. By using the list pointers the addresses of the new array first and last elements can be found. The list first and list last change in a similar manner to the array first and last. When their values are the same (in the case when there is no inherited rotation element), then they change in the same way. When there is a difference their positions change the number of list elements between the rotation element and the array first and last elements is the same.

Unfortunately in a recursively rotated list changing these elements at one level can affect the values at other levels. Therefore a check of the array and list, first and last pointers for all levels in the recursively rotated list is required, as they may need to have their pointers updated. This leads to a worst case running time of $O(h)$ for an easy exchange where h is the level of the recursively rotated list.

Hard exchanges require similar modifications. A hard exchange performed on a level 0 rotated list can affect the pointers, however only in a particular case. If the rotated list's array first and last pointers are not equal to its list first and last, then a hard exchange that occurs before the array first and last elements will cause these to shift by one position. The list first and last elements do not change position. If the lists array and list pointers are equal then this does not happen. The easy exchanges that need to be performed for a recursive hard exchange will change their first and last pointers as described above.

In a standard hard exchange the position to add the new element can be deduced from the order of the elements. This is not possible in a list. Therefore a hard exchange will be given a pointer to the location where the element is to be added. For the initial call of a hard exchange this value will be passed in, for the subsequent recursive calls the previous and next pointers can be used. When performing a recursive hard exchange there are two hard exchanges that need to be performed. The first is at the insertion point where the position is already known. The second occurs after the sequence of easy exchanges. The position for this hard exchange is found by following the next pointer from the last element in the previous recursively rotated list (the list which had an operation performed on it most recently).

An example of a recursive hard exchange in the new structure

The initial recursively rotated list is pictured in Figure 4.11. This rotated list contains the elements **A**...**D**, **F**...**O** in that order. The First and Last tables represent the list first and last pointers that are used in the new structure. Shown are the indices of the first and last elements of the recursively rotated lists of levels 3, 2

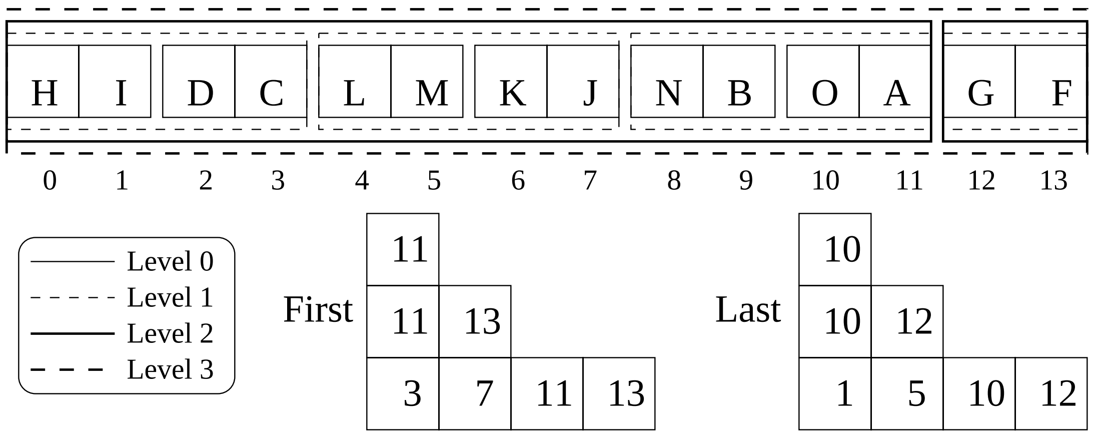

Figure 4.11: The initial recursively rotated list

and 1. Level 0 rotated lists are not pictured as they are indicated by the value of the elements. The array first and last are not shown for brevity's sake, since they are not used for this operation. The one point when they need to be modified it will be mentioned.

To perform a hard exchange in this recursively rotated list, the element to be inserted and the position for the element to be inserted must be known. For this example the element **E** is to be added before the element **F**. The recursively rotated list in this example is of level 3. To perform the recursive hard exchange the level 2 recursively rotated list which contains the insertion point is found. This is done using simple index arithmetic. In this example the second level 2 recursively rotated list contains the insertion point. This process is repeated until the level 0 rotated list containing the insertion point is found. A hard exchange is performed to add the element into the level 0 rotated list. This is shown in Figure 4.12. The element that is removed from the list is **G**.

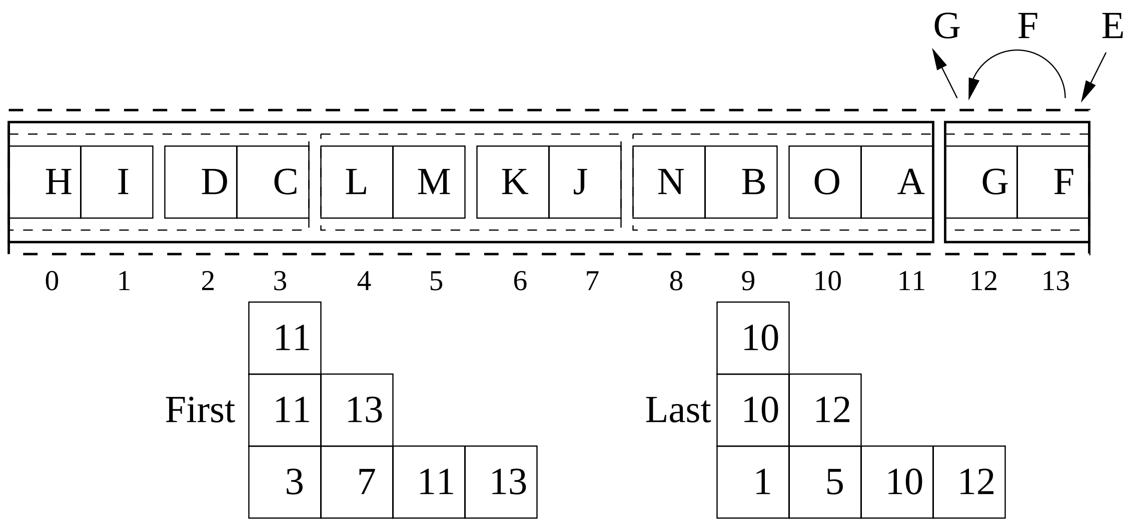

Figure 4.12: The level 2 recursive hard exchange: add **E** and delete **G**

Having just completed a level 0 hard exchange the algorithm must decide if the level 1 hard exchange is complete. To do this it compares the rotation element of the level 0 list to the rotation element of the level 1 list. If they are the same then the level 1 hard exchange is completed. In this case, the rotation element from level 0 is the rotation element from level 1. Notice that the Last table has a 12 in its position for the level 1 rotated list. The element that was in position 12 was **G**. Similarly for level 2, **G** was the rotation element. Therefore this operation also completed the level 2 hard exchange. However **G** is not the last element of the level 3 recursively rotated list (**O** is) so the operation is not complete.

To continue the operation, the next level 2 recursively rotated list is checked. If it does not contain the rotation element of the level 3 recursively rotated list then an easy exchange is performed; if it does, as in this case, a hard exchange is performed. Using the list pointer from **G**'s previous location, the location of **H**, the

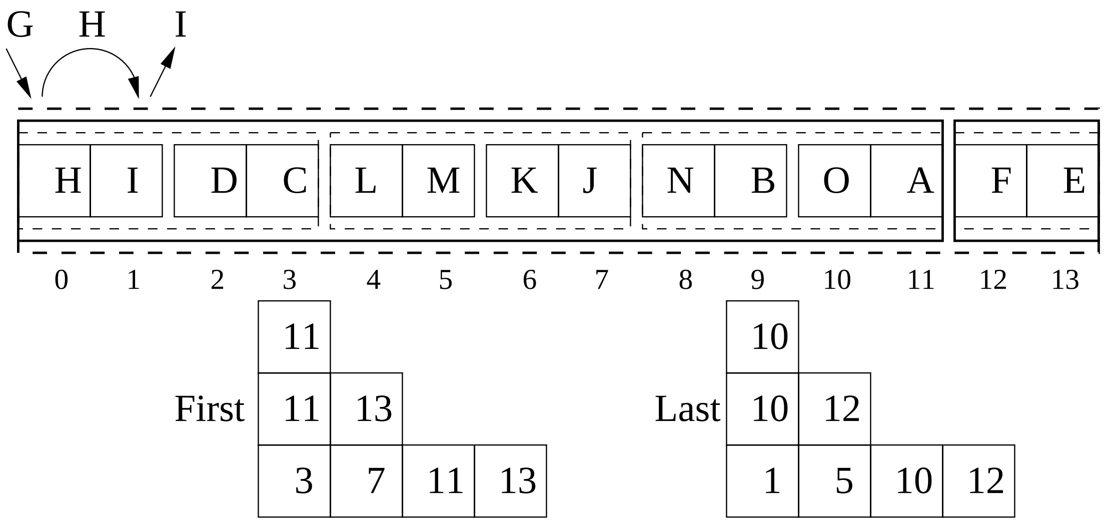

Figure 4.13: A level 1 hard exchange: add **G** and delete **I**

next element in the list, can be found. Using index arithmetic the level 1 recursively rotated list which contains **H** can be found. To perform the hard exchange in the level 1 rotated list the correct level 0 rotated list is determined and a hard exchange is performed in that list. This is shown in Figure 4.13. Since this level 0 rotated list contains the rotation element from the level 1 recursively rotated list, this operation completes the level 1 hard exchange. Had it not contained the rotation element, the next level 0 rotated list would have been checked for the level 1 rotation element. If it did contain the rotation element a hard exchange would be performed, otherwise an easy exchange (Since there are only two level 0 rotated lists per level 1 rotated list this second list would have to contain the rotation element).

Continuing the level 2 hard exchange, the next level 1 rotated list (**L,M,K,J**) is considered. A check for the rotation element (**O**) from the level 2 recursively rotated list is performed. This level 1 rotated list does not contain this rotation element.

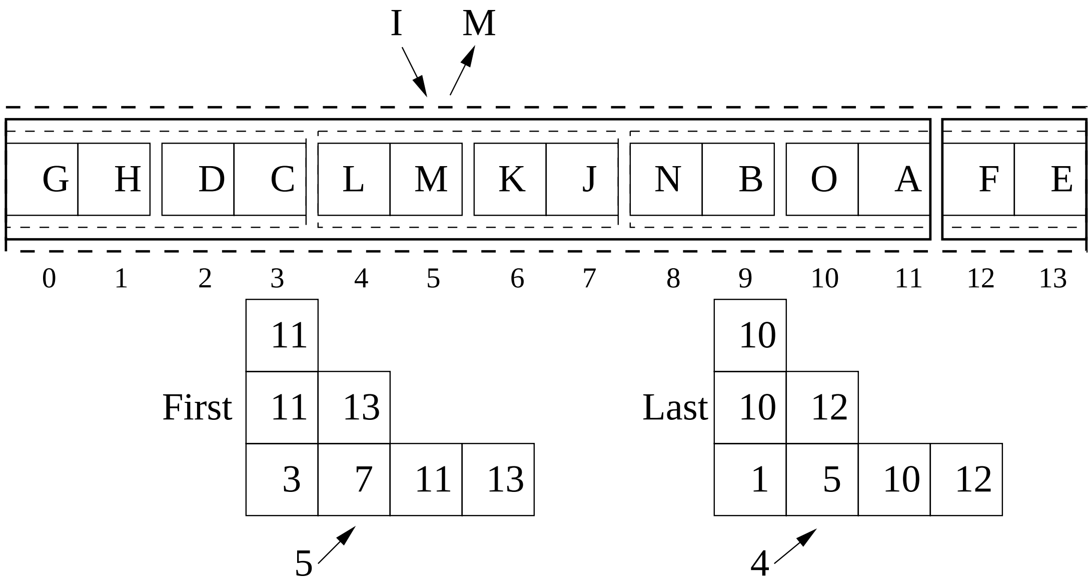

Figure 4.14: A level 1 easy exchange: add **I** and delete **M**

Therefore an easy exchange is performed. The last element in the list is found using the Last table. The element **M** (index 5) is replaced with the shift element, **I**. The First and Last tables are updated, the old Last for this list has become the First and the element second to last is now the last. The array first and last elements are also updated at this point. This operation is shown in Figure 4.14.

The next level 1 list is checked for the last element and it is found within this list. Therefore a recursive hard exchange in performed. Therefore a hard exchange is performed in the level 0 list containing the next element. This is shown in Figure 4.15.

The next level 0 rotated list does contain the last element from the level 1 rotated list so a hard exchange is performed. This is shown in Figure 4.16. The last element in the level 0 list is the last element in the level 1 rotated list so this

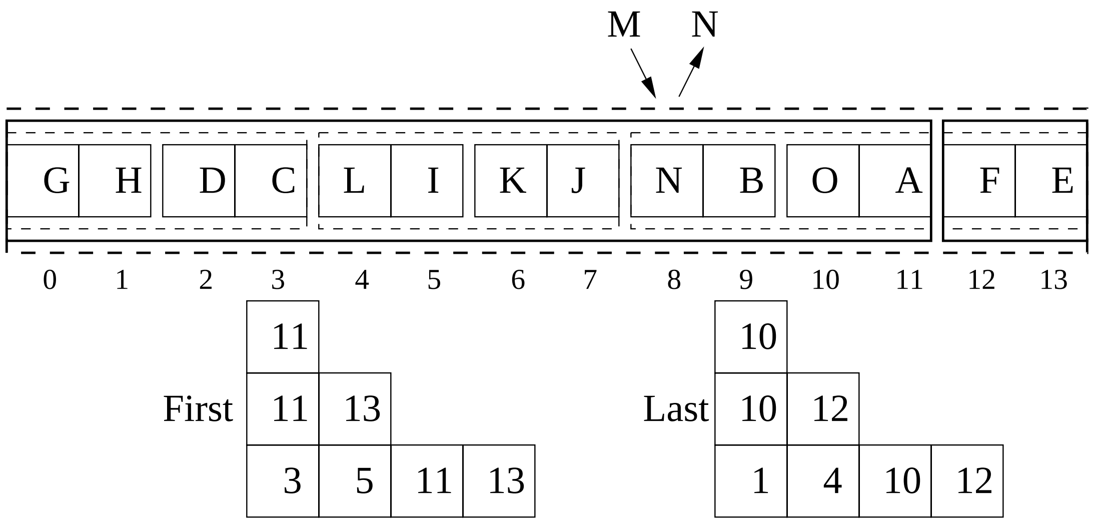

Figure 4.15: A level 0 hard exchange: add **M** and delete **N**

completes the level 1 hard exchange. This also completes the level 2 hard exchange and thus completes the hard exchange in this rotated list. The rotated list which results from this operation is shown in Figure 4.17.

4.3.3 Insertion and Deletion

Insertions use hard and easy exchanges to perform most of the structural modifications. Therefore most of the required changes to the first and last pointers are handled. The changes that are not handled are those caused by the insertion of an element at the end of the array. If the last rotated list has space available then insert the new element into that list. It is possible that this new element is a new first or last element. If it is, then changes to the pointers are required. If i is the level of the lowest level list such that the new element is not a new first or last, then changes need only be made to the level $0, \ldots, i-1$ lists.

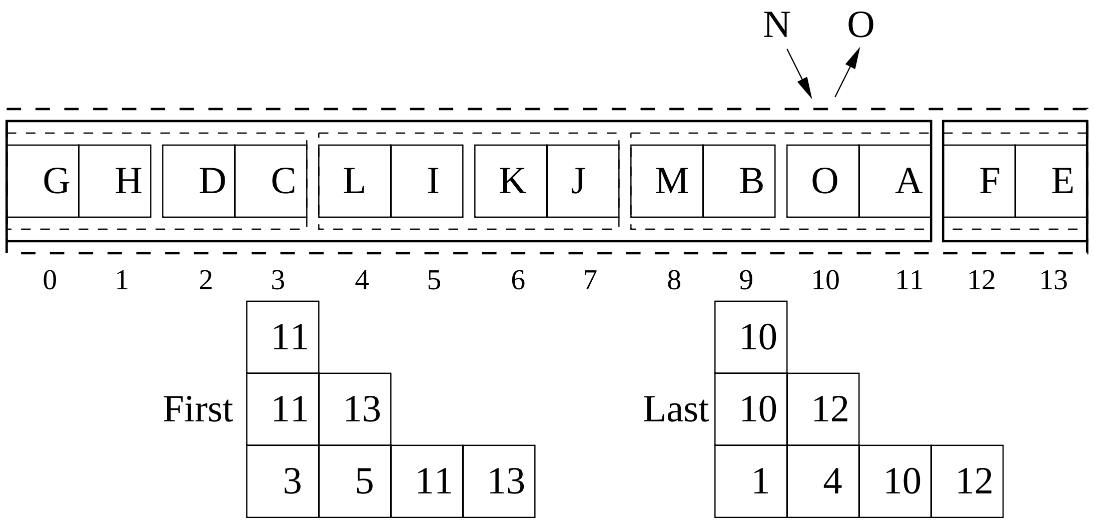

Figure 4.16: A level 0 hard exchange: add **N** and delete **O**

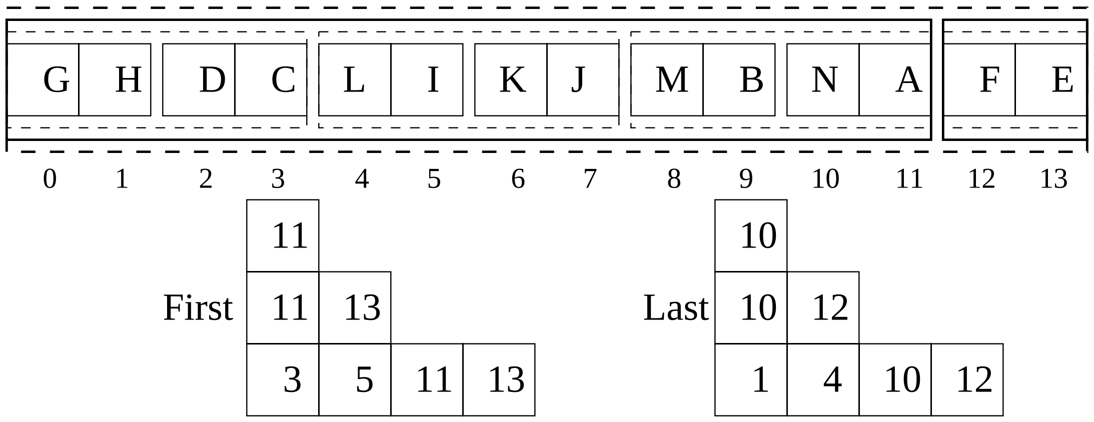

Figure 4.17: The recursively rotated list after the hard exchange.

In the case when one or more new lists are created the new element is clearly the first and last element for all the new recursively rotated lists. For the higher level recursively rotated lists a check is performed, as in the previous case, to see if this element is a new first or last element.

Deletions require modifications similar to insertions. When deleting an element that is the first or last of a recursively rotated list, the list pointers can be followed to find its next or previous element. The associated pointers can then be modified appropriately.

4.3.4 Caching Characteristics

Now consider scanning the list from beginning to end (or from end to beginning). Clearly since this data structure has the list pointers a scan takes $\Theta(n)$ time. More interestingly, consider the caching characteristics of this operation. The following theorem will show that scanning this list causes an asymptotically optimal number of cache misses.

Theorem 10 *A scan of a list stored in this structure incurs* $\Theta(\frac{n}{L})$ *caches misses.*

Proof: Each rotated list represented in this data structure introduces one array last element. Each array last element can cause a constant number of additional cache misses because it represents a discontinuity in the order of the list elements. However the array last element created by rotated lists that are smaller than the length of a cache line do not cause additional cache misses as their discontinuities occur within the data in a cache line.

Let $B(i)$ be the branching factor of level i and assume that $B(i) \geq 2$ for all i. To find the total number of array last elements that occur in a recursively rotated list of n elements use the following observation: There are $\frac{n}{c}$ level 0 rotated lists, each with one last pointer. $B(1)$ of these form a level 1 rotated list each of which has a level 1 pointer, so there are $\frac{n}{c \cdot B(1)}$ level 1 pointers. Similarly there are $\frac{n}{c \cdot B(1) \cdot B(2)}$ level 2 pointers. The total number of pointers in one table is

$$\frac{n}{c} \cdot \sum_{k=1}^{h} \frac{1}{\prod_{i=1}^{k} B(i)}$$

If $L \leq c$ then the total number of array last elements is (within a constant factor) the number of addition cache misses incurred. Since $B(i) \geq 2$, $\prod_{i=1}^{k} B(i) \geq 2^k$. Therefore

$$\begin{aligned}
\frac{n}{c} \cdot \sum_{k=1}^{h} \frac{1}{\prod_{i=1}^{k} B(i)} &\leq \frac{n}{c} \cdot \sum_{k=1}^{h} \frac{1}{2^k} \\
&\leq \frac{n}{c} \cdot (2 - \left(\frac{1}{2}\right)^h) \\
&\leq \frac{n}{c} \cdot (2 - \left(\frac{1}{2}\right)^h) \\
&\leq \frac{n}{L} \cdot (2 - \left(\frac{1}{2}\right)^h)
\end{aligned}$$

thus in $O(\frac{n}{L})$. Otherwise let j an integer such that $c \cdot \prod_{i=1}^{j-1} B(i) < L \leq c \cdot \prod_{i=1}^{j} B(i)$. Then the total number of array last elements that occur in a recursively rotated lists of size greater than L is

$$\frac{n}{c} \cdot \sum_{k=j}^{h} \frac{1}{\prod_{i=1}^{k} B(i)} = \frac{n}{c \cdot \prod_{m=1}^{j} B(m)} \cdot \sum_{k=j}^{h} \frac{1}{\prod_{i=j}^{k} B(i)}$$

$$\begin{aligned}
&= \frac{n}{c \cdot \prod_{m=1}^{j} B(m)} \cdot \sum_{k=0}^{h-j} \frac{1}{\prod_{i=0}^{k} B(i+j)} \\
&\leq \frac{n}{L} \cdot \sum_{k=0}^{h-j} \frac{1}{\prod_{i=0}^{k} B(i+j)} \\
&\leq \frac{n}{L} \cdot \sum_{k=0}^{h-j} \frac{1}{2^k} \\
&\leq \frac{n}{L} \cdot \left(2 - \left(\frac{1}{2}\right)^{h-j}\right)
\end{aligned}$$

which is also $O(\frac{n}{L})$. Since each of these array last elements can introduce at most a constant number of cache misses the number of additional cache misses for a scan in a recursively rotated list is at most $O(\frac{n}{L})$. Therefore the number of caches misses incurred from a scan of this structure is $O(\frac{n}{L})$. It is obvious that a lower bound of $\Omega(\frac{n}{L})$ holds so, a scan of the entire list take $\Theta(\frac{n}{L})$ cache misses. □

This is an optimal result for a scan of the entire list. However it is interesting to consider shorter sequences. Assume $L < c$. The worst case sequence for a level i recursively rotated list is as follows: First a worst case sequence in a level $i-1$ sub-list, followed by a jump from one level $i-1$ sub-list to another (causing a cache miss) and then a worst case sequence in the destination level $i-1$ sub-list. From this simple recursive idea it follows that a worst case a sequence of 2^h cache misses could occur in a row.

4.3.5 Runtime

To determine the running time of an insertion or deletion the branching factor for the structure must be chosen. A good branching factor should be chosen to minimize the running time of the various procedures. To find a good value the following analysis of the running time of a hard exchange will be used.

Lemma 1 *The running time of a hard exchange in a level h recursively rotated list with branching factor $B(j)$ at level j is $R(h) = 2^h(c+h) + h\sum_{j=1}^{h}(B(j)-2)$.*

Proof: Performing a hard exchange in a level 0 recursively rotated list takes $c+h$ time. The c term comes from performing the hard exchange and the h term is the result of having to update pointers. For a level i recursively rotated list perform a hard exchange in the level $i-1$ rotated list in which the element resides, then a sequence of easy exchanges through the rotated lists of the same size and then another hard exchange in the rotated list containing the last element. In the worst case there are $B(i)-2$ rotated lists between the two hard exchanges. Therefore running time, $R(i)$, of a hard exchange in a sub-list of level i is given by the following recursion:

$$\begin{aligned} R(i) &= 2R(i-1) + h(B(i)-2) \\ R(0) &= c+h \end{aligned}$$

Solving this recurrence leads the the following formula

$$R(i) = 2^i(c+h) + h \cdot \sum_{j=1}^{i}(B(j)-2)$$

When $i = h$

$$R(h) = 2^h(c+h) + h \cdot \sum_{j=1}^{h}(B(j)-2)$$

$\square$

To optimize the runtime of a hard exchange the two terms in this this formula need to be balanced. The higher the branching factor the shorter the tree, thus the fewer hard exchanges that need to be executed. However, as the branching factor is increased the number of easy exchanges that need to be executed also increases. In a similar structure Fredrickson uses a branching factor of $B(i) = 2^i$.

Theorem 11 *If $B(i) = 2^1$ then a hard exchange can be performed in $O(n^{\frac{2}{\sqrt{2\log n}}}\sqrt{logn})$ time.*

Proof: Using the branching factor $B(i) = 2^i$ the running time can be computed as follows. First the height of the tree h can be found.

$$\begin{aligned}
n &= c\prod_{i=1}^{h} B(i) \\
&= c\prod_{i=1}^{h} 2^i \\
\log n &= \log c + \sum_{i=1}^{h} i \\
\log n &= \log c + \frac{h(h+1)}{2} \\
h(h+1) &= 2(\log n - \log c) \\
h^2 + h - 2(\log n - \log c) &= 0
\end{aligned}$$

To solve for h, use the quadratic formula. After some simplification, the following equation is obtained.

$$h = \pm\sqrt{2\log n - (\log c - \frac{1}{4})} - \frac{1}{2}$$

Since $h > 0$, the positive root is the desired one.

$$\begin{aligned} h &= \sqrt{2\log n - (\log c - \frac{1}{4})} - \frac{1}{2} \\ &= \sqrt{2\log n} - C \quad \text{for some } C \geq 0 \\ &\leq \sqrt{2\log n} \end{aligned}$$

Using Lemma 1 and the results obtained so far, the running time of a hard exchange using $B(i) = 2^i$ in a level h recursively rotated list is

$$\begin{aligned} R(h) &= 2^h(c+h) + h\sum_{j=1}^{h}(2^j - 2) \\ &= 2^h(c+h) + h(\sum_{j=1}^{h} 2^j - \sum_{k=1}^{h} 2) \\ &= 2^h(c+h) + h(2^{h+1} - 1) + 2h^2 \\ &= 2^h(c+h) + h2^{h+1} - h + 2h^2 \end{aligned}$$

The dominant term of this equation is $h2^{h+1}$. Therefore the running time is $O(n^{\frac{2}{\sqrt{2\log n}}}\sqrt{\log n})$. (This notation is preferred over $O(2^{\sqrt{2\log n}}\sqrt{\log n})$ as it emphasizes the operation is sub-polynomial). $\square$

Using this result the running time for an insertion or deletion can be found.

Theorem 12 *The running time of an insertion or deletion in a list with n elements is $O(n^{\frac{2}{\sqrt{2\log n}}}\sqrt{\log n})$, the same as a hard exchange.*

Proof: The running time for insertion and deletion is very similar. First these algorithms must find the direction to operate, then perform a hard exchange followed by $B(h)-2$ easy exchanges and finally a recursive call with level $h-1$. Therefore the following analysis will apply to both. For convenience insertion will be used.

Let $I(h)$ be the runtime of an insertion in a level h recursively rotated list. A proof by induction will show that $I(h) \leq c \cdot h^2 + R(h)$. The result is then immediate.

A recurrence for the running time of an insertion is

$$\begin{aligned} I(i) &= i + R(i-1) + h(B(i)-2) + I(i-1) \\ I(0) &= c \end{aligned}$$

The base case is $I(0) = c = R(0)$ so the equation holds. Now assume that it is true for level j. Then

$$\begin{aligned} I(j+1) &= j+1+R(j)+h(B(j+1)-2)+I(j) \\ &\leq j+1+R(j)+h(B(j+1)-2)+c\cdot j^2+R(j) \\ &\leq j+1+c\cdot j^2+R(j)+h(B(j+1)-2)+R(j) \end{aligned}$$

From the proof of Lemma 1 the following substitution can be made.

$$j + 1 + c \cdot j^2 + R(j) + h(B(j+1) - 2) + R(j) \quad \leq \quad c \cdot (j+1)^2 + R(j+1)$$

Therefore the running time of insertion or deletion is asymptotically equivalent to the running time of a hard exchange. Thus the running time of insert is $O(n^{\frac{2}{\sqrt{2 \log n}}} \sqrt{\log n})$. $\square$

4.4 Storage Reduction

The recursively rotated list was originally presented as an implicit data structure. That is, a data structure which uses at most a constant amount of extra storage beyond that required to hold the data. The modifications described here add extra storage requirements in the form of pointers. However many of these pointers are unnecessary and can be removed. The additional storage will be reduced to $\frac{8}{c}n$ where c is the size of a level 0 rotated list.

4.4.1 More Modifications

Each node in the list contains the data element and two pointers. These pointers indicate which node is the next or the previous in the list. However the majority of nodes are contiguous in memory, that is the next node or previous node is actually next or previous (in the rotated sense) in the array. Discontinuities occur after an array last element (or before it depending on the direction of the scan). Define a ***jump*** to be the process of moving from a node to the next node when the next node is not contiguous. Since the number of array last elements is low compared to the total number of elements, explicitly storing the implicit relationships between contiguous nodes is extremely wasteful. These $2n$ pointers will be removed from the structure and a method for finding the next or previous node will be given that will not effect the running time or cache complexity of a scan.

Recursively rotated lists allow array last elements to occur in arbitrary locations, thus any node in the list is potentially an array last element. The locations of the array last elements are stored in the array last table. These values can be used to find the location of jumps. Using the level of the jump the destination recursively rotated list can be found. The first element within the destination recursively rotated list can be obtained using the array first table.

The element after (in list order) the array last element of a level i recursively rotated list is in the next recursively rotated list of level i, within the level $i + 1$ list. Therefore the list first element of the next recursively rotated list is the next node in the list. It is important to remember that the concept of being next in a recursively rotated list includes the possibility of wrapping around to the first sub-list of the level $i + 1$ recursively rotated list.

An example of navigation

The rotated list in Figure 4.18 will be used as an example of the navigational functions. The entire rotated list stores the elements from **A** to **Y**, although only the first level 2 recursively rotated list is pictured. It stores the elements **A** to **F** and **T** to **Y**. The array first and last elements are indicated for this section of the recursively rotated list in two tables. Figure 4.18a shows the unrotated version

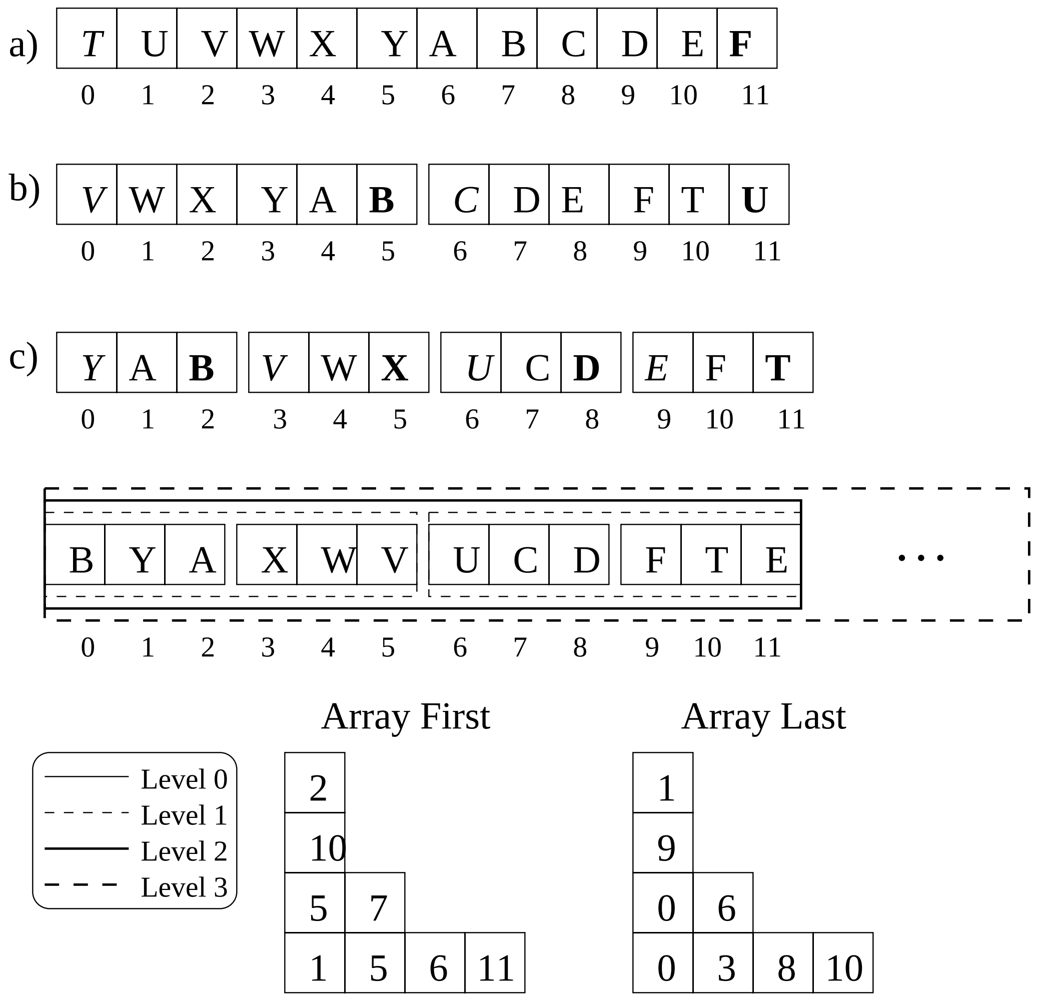

Figure 4.18: The recursively rotated list to be walked through.

of the level 2 list. The array first element is shown in italics and the array last element is shown in bold. Figure 4.18b and 4.18c show the level 1 and 0 sub-lists respectively. The final position of these element in the complete rotated list is given in the tables.

To walk through this list, begin at element **A**. Element **A**'s index is 2 which is not an array last element. This means that the next element is the next element in the level 0 rotated list. Since this is a rotated list the next element is **B**. Element **B** is in the array last table at level 0 and level 1. Using the largest level, the element **B** has a level 1 jump. This jump goes to the next level 1 rotated list, which is the second 6 element rotated list. From the array first table look up the array first element for this rotated list. The table gives index 7 which corresponds to element **C**. This element does not appear in the array last table so it does not represent a jump. The next element is **D** which does appear in the array last table. There is a level 0 jump from **D** to **E** (index 11 in the array first table). Element **E** to **F** is a simple step. Element **F** is the array last element for the level 2 list. From **F** the jump goes to the neighbouring level 2 recursively rotated list (which is not shown here). When returning from walking the neighbouring recursively rotated list the algorithm looks in the array first table for the level 2 list and gets index 10 which corresponds to element **T**. Element **T** is a level 0 jump to element **U**. Element **U** is a level 1 jump to element to **V**. Elements **V**, **W** and **X** are all in the same level 0 list. Element **X** is a level 0 jump to element **Y**.

How can this procedure be performed efficiently? When stepping through a level h recursively rotated list, two arrays of pointers, both of length h will be maintained. The locations of the array last elements of all the recursively rotated lists which contain the current position is stored in one array. The other array points to the elements after the jumps. The order of the elements within the array is the order that the array last elements appear physically in memory. A pointer into this array indicates the position of the next jump. If the current element is not the array last element indicated by the pointer then the next element is the next element in memory. If it is the array last element, then the next element is found by following the jump.

After following a jump these arrays must be updated to reflect the new array lasts. Assume the algorithm has just followed a level i jump. Then the pointers associated with the jumps of levels less than or equal to i have all changed value. The smallest and largest of these pointers define a range within the array. All the pointers that reside within this range may need to moved. So the time of a single jump is proportional to the length of time required to sort the pointers within this range.

For this method to be useful, the running time of a scan using this modified structure must still be $\Theta(n)$ and it must incur $\Theta(\frac{n}{L})$ caches misses. The next two theorems will show these modifications can be used without seriously affecting the running time or caching characteristics of a scan.

Theorem 13 *Performing a scan of this modified list takes* $\Theta(n)$ *time.*

Proof: To analyse the running time consider the following cost model. When updating an array assume that the number of element to be modified is the highest level of a jump within the level 0 rotated list. Assume that the time required to re-arrange these elements in the square of the number of elements and charge this cost for every step in the level 0 rotated list, not just when following a jump. The model is clearly far more expensive than the actual cost and is an upper bound on the actual cost for any recursively rotated list.

A worst case for this model can be found by trying to maximize the number of level 0 rotated lists with large jumps. To do this consider only the jumps of level greater than 0 since every level 0 rotated list contains one level 0 jump. The obvious thing to do is to put one high level jump per level 0 rotated list, making the number of high level jumps as large as possible. This is a worst case in the given cost model, but is it a worst case for the actual case? It may not be, however the cost generated using this model and this example will be an upper bound on the running time of any example in the actual cost model. This example provides the maximum number of expensive operations.

To begin the analysis define d_i to be the number of jumps of level i. The value of d_i can be found by noticing that each rotated list of level i contains one level i jump. The number of level i rotated lists is $\prod_{j=i+1}^{h} B(j)$ since each level j rotated list contains $B(j)$ level $j-1$ rotated lists and there is one level h rotated list.

As discussed above, for each level 0 list containing a level j jump charge $c \cdot j^2$. The total charge for the entire list is $c \cdot \sum_{i=1}^{h} d_i i^2$. Since $n = c\prod_{i=1}^{h} B(i)$,

$$\begin{aligned} \frac{d_i}{n} &= \frac{\prod_{j=i+1}^{h} B(j)}{c\prod_{k=1}^{h} B(k)} \\ &= \frac{1}{c\prod_{k=1}^{i} B(k)} \end{aligned}$$

So the cost equation simplifies to the following:

$$\begin{aligned} cn \cdot \frac{\sum_{i=1}^{h} d_i i^2}{n} &= cn \cdot \sum_{i=1}^{h} \frac{d_i i^2}{n} \\ &= n \cdot \sum_{i=1}^{h} \frac{i^2}{\prod_{k=1}^{i} B(k)} \end{aligned}$$

As before, $B(i) \geq 2$ so the following inequality holds

$$n \cdot \sum_{i=1}^{h} \frac{i^2}{\prod_{k=1}^{i} B(k)} \leq n \cdot \sum_{i=1}^{h} \frac{i^2}{2^i}$$

Since the sum in the final equation converges this value is in $O(n)$. Therefore the total time for a scan of the list is $\Theta(n)$. $\square$

Theorem 14 *The number of cache misses incurred in a scan of the modified list is in* $\Theta(\frac{n}{L})$.

Proof: Now to see that the number of additional cache misses is $O(\frac{n}{L})$, consider the following caching method. Reserve three cache lines for storing the L elements on either side of the pointer into the tables. When a jump whose range falls within these L element is performed, the elements can be sorted using bubble sort without any additional cache misses. When a jump is followed that is outside of these $2L$ elements, the elements are sorted using bubble sort and incur i^2/L cache misses, where i is the number of elements to be sorted.

The model and example used in the proof of Theorem 13 will again be utilized to bound the worst case. As before the size i of a jump will be determined by the largest element in the range of elements to be sorted. If $i \leq L$ then there is no charge for the sort, otherwise charge i^2/L. The number of additional cache misses incurred by this method is then bounded by:

$$\begin{aligned}
\sum_{i>L} d_i \cdot \frac{i^2}{L} &= \sum_{i>L} \prod_{j=i+1}^{h} B(j)\frac{i^2}{L} \\
&\leq \sum_{i>L} 2^{h-i}\frac{i^2}{L} \\
&\leq \sum_{i>L} \frac{2^h}{L}\frac{i^2}{2^i} \\
&\leq \sum_{i>L} \frac{2^h}{L}\frac{i^2}{2^i}
\end{aligned}$$

This sum can be bounded by twice the first term in the summation. This leads to the following equation

$$\begin{aligned}
\sum_{i>L} \frac{2^h}{L}\frac{i^2}{2^i} &\leq 2 \cdot \frac{2^h}{L} \cdot \frac{L^2}{2^L} \\
&\leq 2 \cdot 2^h \cdot \frac{L}{2^L}
\end{aligned}$$

This is in $O(\frac{n}{L})$. Therefore this method introduces at most $O(\frac{n}{L})$ additional cache misses to a scan of the structure. □

4.4.2 Storage Requirements

This modification removes the $2n$ pointers associated with the list nodes. The only additional storage beyond the n locations required for the data are the four arrays required for storing the array and list, first and last pointers. The number of pointers in one of these arrays is:

$$\begin{aligned}\frac{n}{c}\sum_{i=1}^{h}\frac{1}{\prod_{j=1}^{i}B(j)} &\leq \frac{n}{c}\sum_{i=1}^{h}\frac{1}{2^i} \\ &\leq 2\cdot\frac{n}{c}\end{aligned}$$

Four arrays of this size are required, so the total amount of storage is less than $\frac{8n}{c}$. If c is picked to be larger than 8, then the amount of extra storage is less that n. In fact c can be chosen as large as desired to reduce the extra storage as long as n is sufficiently large. Instead of the $2n + \frac{8n}{c}$ space that was required before this additional modification, only $\frac{8n}{c}$ additional pointers are required.

4.4.3 A Drawback

There is one significant drawback to this modification. Although a scan can be performed from an arbitrary position in the list, there is an associated cost that must be paid. Given an arbitrary location in the list it takes $\Theta(h)$ time and cache misses, where h is the recursive level of the list, to collect the information required to make the first step in the structure. Calculations must be performed to find the rotated lists in which the current position resides. As well the values of their last pointers must be copied into the array of jump pointers. The locations these values are read from are independent locations in memory so each will cause a cache miss.

4.5 The Final Structure

The recursively rotated list can improved further by using Fredrickson's method.

Theorem 15 *A structure can be built that implements a linear list with insertion and deletion times in* $O(n^{\frac{1}{\sqrt{\log n}}}\sqrt{\log n})$*, and allows scans of the entire list taking* $\Theta(n)$ *time and incurring* $\Theta(\frac{n}{L})$ *cache misses.*

Proof: Let $f(i) = 2^{i^2}$ and let the sub-arrays be represented as level i recursively rotated lists with $B(i) = 2^i$. The number of sub-arrays r is approximately $\sqrt{\log n}$. Therefore the largest sub-array is a level r recursively rotated list. An insertion (or equivalently a deletion) is performed by executing a hard exchange in the sub-array in which the insertion occurs, a series of easy exchanges until the last recursively rotated list is reached and an insertion is performed in that list. The running time of an insertion or deletion is dominated by the running time of the operation in the largest sub-array. Therefore these operations take $O(n^{\frac{1}{\sqrt{\log n}}}\sqrt{\log n})$ time.

The caching characters of a scan is basically the sum of the scans in each of the recursively rotated lists. The only added complication is the movement between lists. However these misses are easily shown to be insignificant. If $n \leq L$, then they do not cost anything as the jump stays within the cache line. When $L < n$ each such jump is the result of having scanned through a recursively rotated list who length is greater than L. Therefore the frequency of such misses is less than $O(\frac{n}{L})$ and so do not affect the overall cache complexity in a negative fashion. $\square$

4.5.1 Memory Management

The discussion of memory management has been delayed to this point because knowledge of the final structure and the running time of its operations can be used to reduce wasted space. Notice that the structure is stored as a set of arrays.

For a structure to grow new arrays must be allocated and others must be resized. There are many resizeable array data structures, [BCD$^+$99, GK99]. However, these simulate an array using multiple sub-arrays. These solutions are undesirable for cache oblivious data structures because moving between the sub-arrays causes cache misses. Each of these sub-arrays will introduce an additional cache miss for a scan. Instead of a scan through a resizeable array taking $\frac{n}{L} + 1$ cache misses, it will take $O(\frac{n}{L} + f(n))$. The $f(n)$ term is relatively small, $\sqrt{n}$ in [BCD$^+$99], however it interferes with the optimal performance. Therefore an actual array will be used to store the data. Resizing these arrays will be done using dynamic tables [CLR91].

Notice that for this structure constant time insertion or deletion is not required. In fact, these operations are super logarithmic, so having an amortized cost of $O(\log n)$ is acceptable. Why is this desirable? By re-allocating the arrays to size $n + \frac{n}{\log n}$, the unused space is in $o(n)$.

Memory management will work as follows: The array which stores the nodes of the list will be used to determine when re-allocations are required. The other arrays will be resized or allocated so that they are able to support the array of nodes if it becomes full. When the array of size n is full it will be re-allocated to be of size $n + \frac{n}{\log n}$. If the array contains less than $n - 2\frac{n}{\log n}$ elements it will be shrunk to size $n - \frac{n}{\log n}$. The resizing and copying of arrays will be done in $O(n)$ time. Clearly there will be at least $\frac{n}{\log n}$ operations between each resizing so the amortized cost of these resizing operations is $O(\log n)$. Therefore the amortized cost of adding or removing elements from the array is insignificant in comparison to the running time of an insertion or deletion.

Chapter 5

Future Work

This thesis takes a few simple abstract data types and shows how they can be implemented in a cache oblivious fashion. There are many other data types that could be studied in a similar manner. Hopefully, the work presented here contains some ideas that will be useful for future research in these areas. Alternatively, the data structures presented here could be used as components to implement other data types to improve their caching characteristics.

The direct practical applications of this work is limited. Although guaranteeing optimal caching results, the impact of the improvement may be offset by the increase in running time of the operations. The caching environment, specifically the size of a cache line and the cache miss penalty, will determine if these structures give an actual improvement. A scan of the list structure is probably faster than a scan of a linked list when run in main memory and definitely faster when going to disk. However the complexity of the structure and the relatively poor insertion and deletion times make this structure undesirable. A search in the tree layout is probably competitive to a standard binary search when run in main memory. The increased running time of the search will offset the gains from the improved caching. When going to disk the reduction in cache misses will give greater improvements and the new structure will be faster.

Frigo et al.[FLPR99, Pro99] give two directions for future work in cache oblivious algorithms. These ideas can also be investigated from a data structure point of view. The first suggestion is called separation. ***Separation*** is studying the difference in cache usage between the best oblivious solution and the best aware solution. Can any data structure be made oblivious without an asymptotically significant caching penalty? The answer would seem to be no, as aware structures can exploit their knowledge of the cache. A proof of this would be very interesting.

The second area they suggest is called simulation. ***Simulation*** compares the efficiency of cache oblivious data structure to cache aware ones. What is the loss in efficiency when making a data structure oblivious? Is there a provable bound on how much worse the running time of operations should be? They suggest using simulation methods to convert cache aware algorithms into cache oblivious ones.

Extending the work presented here is also an interesting direction. Can cache oblivious lists be found that have faster insertion and deletion times? One idea that was considered, but not investigated, was to attempt to amortize the cost of reorganizing the data structure. For example, a list that has its nodes stored in order in an array can have $O(\frac{n}{L})$ cache misses introduced without asymptotically effecting the caching characteristics. If each operation introduces a constant number of cache misses then $O(\frac{n}{L})$ operations can be performed before the structure needs to be reorganized. Unfortunately the number of operations that can be performed before reorganization is required must not depend on L, otherwise the structure becomes cache aware. Is there a method for deciding when to reorganize that does not depend on L?

Another similar idea is to spend some time reorganizing the list every time a data structure operation is performed. For example if every insert spends $O(\log n)$ time reorganizing then maybe the list can be maintained ordered enough to still obtain optimal caching characteristics for a scan. The running time of the operation would be dominated by the amount of time reorganizing the structure, therefore studying how little reorganization is necessary to maintain the optimal caching characteristics would be important. It seems that more than a constant amount of reorganization is necessary, but how much more? Does anything in $\omega(1)$ work or does it actually require a significant amount of work?

Another data structure that would be very useful would be a cache oblivious resizeable array. A data structure that supported constant time insertions and deletions, scans in $O(\frac{n}{L})$ cache misses and which does not waste an excessive about of memory is very desirable. The simple dynamic table used in Section 4 has a tradeoff between running time and wasted space. The manner in which it was used in Section 4 allowed for logarithmic running time, however in many cases this will not be sufficient.

Bibliography

[BCD$^+$99] A. Brodnik, S. Carlsson, E. Demaine, J. I. Munro, and R. Sedgewick. Resizeable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures, (WADS) 99*, volume 1663 of LNCS, pages 37–48. Springer-Verlag, 1999.

[BDFC00] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache oblivious B-Trees. In *The Proceedings of the 41th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 399–409, 2000.

[BFP$^+$73] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[BM72] R. Bayer and E. McCreight. Originization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[CLR91] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1991.

[FLPR99] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *The Proceeding of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–297, 1999.

[Fre83] G. Frederickson. Implicit data structures for the dictionary problem. *Journal of the ACM*, 30:80–94, 1983.

[GK99] M. Goodrich and J. Kloss. Tiered vectors: Efficient dynamic arrays for rank-based sequences. In *The Proceeding of the 6th International Workshop (WADS)*, volume 1663 of LNCS, pages 205–216. Springer-Verlag, 1999.

[GvL89] G. Golub and C. van Loan. *Matrix Computations 2nd Edition*. The Johns Hopkins University Press, Baltimore, 1989.

[LD91] H. R. Lewis and L. Denenberg. *Data Structures and their Algorithms*. Harper Collins, New York, 1991.

[MS80] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21:236–250, 1980.

[PH94] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, San Francisco, 1994.

[Pro99] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[vEB75] P. van Emde Boas. Preserving order in a forest in less that logarithmic time. In *The Proceedings of the 16th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[vEBKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10(2):99–127, 1977.