City University

MSc in Software Engineering

Project Report

2008

Implementation of a Finger Tree in C++

Louis-Henri Freeman Wilcox

Supervised by: Ross Patterson

29-09-2008

Abstract

This work explores data structures, specifically *finger trees,* with an implementation of one in C++. The project will build on the data structure defined in other work and explore whether it is possible to achieve the expected operational efficiency.

Table of Contents

1. Introduction

The aim of this project is to implement a data structure proposed by Hinze & Patterson (2006), called a *finger tree,* in C++. This structure provides the storage of a sequence of data which is used frequently by developers. The finger tree also should provide efficient operational times for a wide variety of common procedures which are used by developers.

The successful implementation of this data structure in C++ will provide these common operations and show how they perform in an efficient manner. This project will explore programming in C++ and whether in this language, it is possible to implement a finger tree as efficiently as it is described by its developers.

This project was chosen due to a passionate interest in C++ and low level programming. The expectation is that by exploring this data structure a beneficial and informative learning experience will be achieved.

2. Literature Review

The finger tree incorporates many different concepts. An understanding of Tree structures, amortising of the cost of operations and persistence are required before the development of a finger tree is pursued.

2.1 Data Structures

It is a common requirement for a program to store and operate on some data. When the amount of data which needs to be stored is unknown and cannot be determined until run time, then a method to store the required data without a predetermined, *bounded* space in memory is required so that memory usage can dynamically increase or decrease when necessary.

The C++ Standard Template Library (STL) provides containers to accommodate these needs as well as algorithms to support them (Stroustrup 1997, p.66). One of the simplest methods provided is a list "The functional programmers favourite data type" (Hinze & Patterson 2006, p.1). A list stores a data item along with the memory address of the next data item in the list, the combination of this information is stored in a *node*. In this way items can simply be added to the end of the list so that it may grow to accommodate all the data required for a running program.

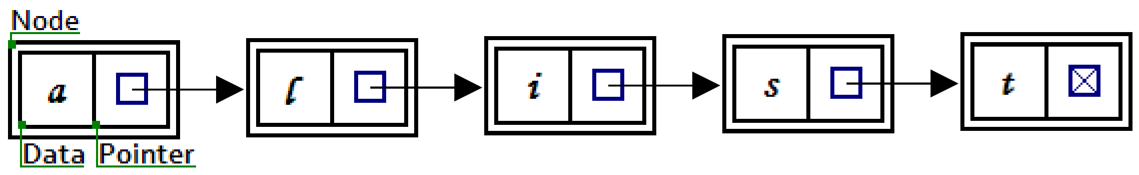


figure 2.1 A simple list data structure

Although a linked list may be the first choice of a programmer, there are many common operations carried out on sequences of data where the performance of a list may be insufficient. For example locating a data item in a list, even if its position in the sequence is known, requires traversal of each node starting from the beginning of the list, or sometimes the end, so that the memory address of where the data is stored can be determined. When the efficiency of performing these basic operations is poor the overall performance of an application also suffers.

Some common operations on containers include adding and removing elements from either the front or the back of the list, concatenation, insertion and removal of an element from any point within the list, searching for an element within a list or splitting the list into smaller lists (Hinze & Patterson 2006, p. 1). While some data structures are specialised to perform extremely well with some of these operations, they often cannot perform all of these functions with equivalent levels of efficiency.

2.2 Time Complexity

Measuring the efficiency of an operation can be done in a literal sense by saying it takes a certain amount of seconds. However this would usually be a meaningless representation for all possible permutations of a functions use and the diverse situations it may be used in. It is more useful to express the time an operation will take on the key factors which will affect it, which could be expressed as a function showing the total number of commands which must be executed based on these factors. One of these effecting factors could be the size of the data set and the amount of operations which must be performed on it. This would provide a function from which the operational time could be established. Additionally, any command an operation must perform which is independent of the size of the data set could be included in a calculation of the operational time as additional constant factors.

Asymptotic notation hides any constant factors that affect the time of an operation i.e. those which are not dependant on the size of the data (Heileman 1996, p.33). Effectively this means any operations which are performed the same number of times, no matter what the size of the data set, are considered to not adversely effect the functions operational time. These usually include statements that do not take place within any loops or make calls to other functions (Kingston 1998, p.29).

A common asymptotic expression of an operation's execution time is the "Big-O" notation, which provides an approximation for the time a given operation will take. An operation's performance represented through the Big-O notation is dependant on the size of data which it must operate on. Big-O shows how the time taken for an operation will grow in relation to the amount of data. For example, the time taken to search through a list is dependant on its length (n). This would be expressed as O(n) in the Big-O notation and this is also known as an operation which takes linear time. While linear operations can be distinguished this way it is possible to express different execution times with a clear indication of how these times will grow in proportion to the size of the data operated on.

Notation	**Name**	**Description**
O(1)	constant	Operation growth is unaffected by size of a data set.
O(n)	linear	Growth is proportional to size of the data set.
$O(n^2)$	quadratic	Growth is proportional to the square of the data set size, for example the processing of each element in the data set requires a pass through the data set itself (embedded for loop.)
$O(2^n)$	exponential	Operational growth is exponential.
O(log n)	logarithmic	These usually imply that the data required to complete the operation can be reduced as it is processed i.e. once one data item is processed the remaining data to be processed may be reduced by half. (divide and conquer)
O(n log n)	log linear	

Big-O notation is used to define the upper bound of an operations' time complexity, this means that even if there are occurrences of a call to the operation which may take place in more efficient bounds, these are not considered in the notation and only the greatest is represented. Although Big-O is the most commonly used notation there are other notations which can be used to show concisely other bounds of an operation. Ω, Big-Omega, notation is used to represent the lower bounds of an operations' time complexity, It shows the least amount of time an operation can take and although it may take longer, it will always be at least the amount shown in the Big-Omega notation (Kingston 1998, p.30). Big-Theta Θ incorporates the worst case and best case running time and therefore states that all operations are bound by this notation.

2.3 Amortisation

The time costs of an operation are usually quite easily determined as the operation usually performs the same set of actions, e.g. the code executes a for-loop through n elements therefore the cost is O(n). However in some cases an operation may perform poorly under some conditions but well under others, thereby having an upper and lower bound. This could occur when an operation executes differently by using 'if else' clauses. Amortisation allows the averaging of the cost of an operation over a sequence of calls instead of considering a single worst or best case operation call. In this way allowances can be made for some operations to have occasional costs which exceed the normal amortized cost, these are known as expensive operations. Similarly there may be operations which are more efficient than the amortised cost. These are conversely called cheap. (Okasaki 1998, p.14) finger Tree operations employ amortised cost so that the data structure can provide efficiency across the many different operations it provides, performing well in all of them unlike other structures.

2.4 Trees

The issues regarding operational efficiency has brought about the development of alternatives for the organisation of data. Although not offering a solution to all the common sequence operations previously stated, Tree structures do enable faster searching by ordering the data by their value and storing any new items at locations appropriate for their value.

Horowitz et al. (1997, p. 74) gives a definition of a general tree structure as

"*A tree is a finite set of one or more nodes such that there is a specially designated node called the root and the remaining nodes are partitioned into n≥0 disjoint sets $T_1,\dots,T_n$, where each of these sets is a tree. The sets $T_1,\dots,T_n$ are called subtrees of the root.*"

The above describes a tree structure as being made up of nodes which still contain data items but these nodes form a tree structure by containing or addressing 0 or more nodes as their children which in turn can address further descendants. The idioms of 'parent' and 'child' are commonly used to describe the hierarchy of the tree and its nodes.

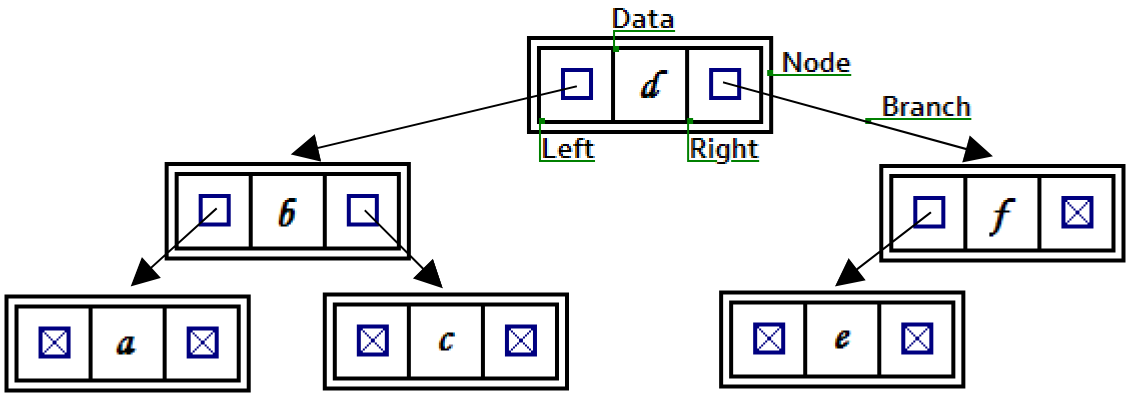


figure 2.2 a simple tree structure

Figure 2.2 demonstrates a simple tree structure, storing characters. The *root* drawn at the top, here storing the character d, and the descendants of the root labelled as left and right branches are also trees. The root node in this diagram has two subtrees or children descending from it. This number of children can be called the degree, thus the root here has a degree of two. A node with a degree of zero can also be called a leaf or terminal node, like those shown at the bottom of the structure. With the use of comparison operators a tree will usually store new items with a lesser value than that contained in the current node on the left branch and those with a higher value on the right. By doing this a tree can locate an item by traversing down the left and right branches using the value of the item to indicate the path to the correct node. As a result, the time taken to search through such a tree is dramatically reduced i.e. at each node traversal, half the tree can be eliminated from the search, resulting in an operational time of O(log n) (Heileman 1996, p.193).

2.5 Finger Tree

A finger tree provides a data structure that can deal with sequences, such as those commonly stored in a list, but it aims to provide good performance for many common operations rather than a subset of them. By using a tree-like organisation, improved efficiency can be achieved when considering the amortised cost of common sequence operations.

Hinze & Patterson (2006, p. 3) adapt a 2-3 tree to only store data in the leaves of the tree as opposed to the tree demonstrated previously, where each node stores data along with subtrees. This enables the storing of a sequence from left to right as demonstrated in figure 2.3. However, using this structure would result in operations to access the data taking a time which is dependent on tree size.

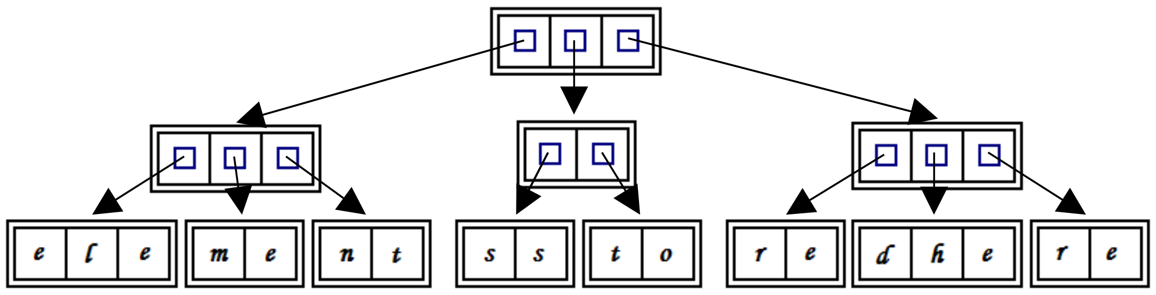


figure 2.3 a Finger 2-3 tree

By providing better access to nodes close to a specific location in a tree the structure becomes a *finger.* A finger Tree employs two fingers to hold a maximum of four nodes from the front and back of the sequence. This supplies constant time access to these ends of the sequence. These two fingers make up a full finger tree node. The full structure of a finger tree is a repeating pattern of this top layer of two fingers. The top layer holds data items and a connection to the next level through a *spine.* The next layer holds nodes of two to three items and each consecutive layer stores nodes of the same type which the previous layer holds, creating the previously described 2-3 trees with their data items stored on the leaves.

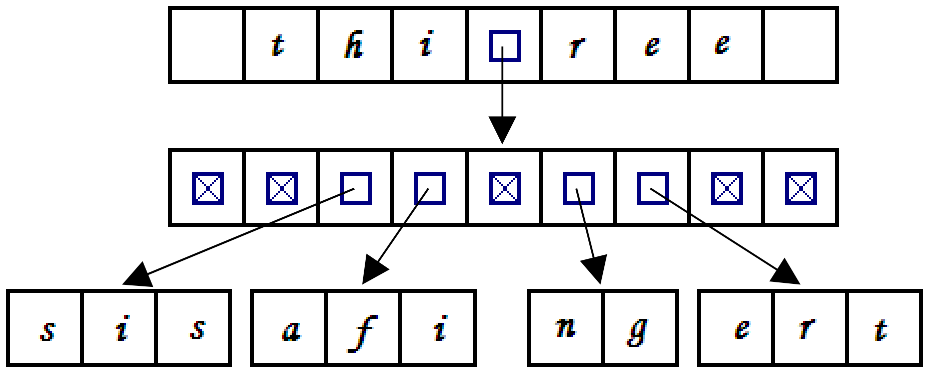


figure 2.4 A Finger Tree of two levels

Figure 2.4 demonstrates a finger tree of two layers. Organised tree structures store items of a lesser value on the left hand side of the tree. Similarly a finger tree stores values closer to the front of a sequence on the left and those at the end on the right. The finger tree node also stores the address of the next level and as the diagram demonstrates this can be considered the middle of the sequence.

This repeated use of the same basic structure at each level enables an implementation to use recursion to manage lower levels. Each level will operate in the same way, so the same functions can be called when it becomes necessary to operate on lower levels.

The tree can be shown to be efficient when adding elements to the front and back of the sequence. As stated the first and last elements are held at the top level, therefore items can be added to this level in constant time. When this level becomes full, items are removed from the top and pushed down the spine onto the level below. These operations could take "Θ(log n) time in the worst case" (Hinze & Patterson 2006, p. 7) but because this occurs after a series of cheaper operations of adding to the top level, the amortised cost is much less at O(1). Other operations, such as concatenation and splitting also perform well, due to the layout of the structure.

2.6 Persistent Data Structures

It is also possible to implement a finger tree as a persistent data structure. Kaplan (2008, p.1.1) states “A data structure is called persistent if it supports access to all versions” i.e. if a data structure is updated, it can be considered fully persistent if the version before the update can still be accessed and altered. Persistence can be implemented by copying the entire structure on an update, but this is very inefficient. A better way is to share data that is common to other versions of the structure so that only areas of difference require copying. A finger tree can achieve efficient persistence due to the large data storage it uses for the left and right fingers. With these, many actions can take place before any copying of data is required to maintain persistence (Kaplan 2008, p.1.2). Efficient persistent data structures provide many benefits to functional programs where many versions of the same data may be in use by multiple parties at any one time.

3. Implementation

This section will explore the implementation of a finger tree in C++.

3.1 Basic Finger Tree Structure

The first step for the finger tree development in C++ is to introduce all the storage elements that will be required to build a finger tree structure.

As described previously, two fingers are required to store, at most, four elements from the front and back of the sequence. All access to data within the structure will be achieved through these areas of storage, either at the top level or all subsequent levels in the tree. As a result access should be as fast and simple as possible to reduce operational time. As the maximum number of elements will always be four, an array will easily provide fast random access. However as these arrays may not at all times be full, counters will also be required to keep track of the number of elements these arrays contain, this will prevent access to parts of the array which have not yet had an element inserted.

The next part of the finger tree data structure will be a connection to its subtrees. These will essentially be repetitions of the top level but contain 2-3 trees of increasing complexity at each subsequent level of the tree. It is important to remember however that these 2-3 trees will only store data items on their leaves. This part of the structure presents issues which will need to be resolved before a full finger tree structure can be realised.

Firstly, it must be established how these 2-3 trees will be stored. This can be resolved by creating an additional structure. One must consider this part of the structure at a hierarchical level. If this new structure has only one level it will hold two to three elements of the required type. This can be called a node of type T, where T is a type of data that needs to be stored and as before, an array will provide the best access to these elements. When two levels are required, the bottom layer will continue to hold elements of type T, while the top layer will store nodes of type T. This is repeated as the node tree increases in height, the bottom layer or leaves always being nodes of type T.

The need to implement a class which can store and operate on different types, can be resolved by writing a generic class which C++ accommodates for with templates.

```
template <typename T> class Node;
```

The statement above will allow the definition of nodes of type int, `Node <int>`, but also could allow the following statement, which describes a tree of two levels.

```
Node < Node <int> >
```

The creation of a generic node container class allows a tree to be constructed by having node containers store nodes of another type. The above statement would create the nodes containing elements of type int, the leaves, and store them in another node. This top node in turn could be stored with others of the same type in another higher level node.

```
template  <typename T>
class Node
{
    private:
    T *contents;
    unsigned int size;

    public:
    Node():size(1){contents=new T[1];}
    Node(unsigned int sz):size(sz){contents=new T[size];}
    ~Node(){delete[] contents;}

    T &operator[](unsigned int i) {return contents[i];}
    T operator[](unsigned int i)const {return contents[i];}
};
```

In this code, overloadings of the array accessor parenthesis, operator ([ ]), have been provided to allow easy access to the elements within the node. This way an object of this class can be used in the same manner as an array.

This use of generic types can be expanded to the overall finger tree structure. This will allow the end user of the sequence container to use it with any type they wish. This generalising will also allow the application of the finger Tree's spine using the same method already used in the node class. By storing in the top level of the finger tree of type T, another finger tree which stores nodes of type T, the finger tree structures basic storage requirements are met. These are two Fingers and a spine of Finger Trees which instead hold node trees of increasing complexity at each level.

The following code conceptually shows how the finger tree is realised from the previous descriptions, however there are complexities with using template parameters in C++ that when examined will require changes to this code.

```
//this demonstrates the storage requirements of a Finger
//however it will not compile
template <typename T>
class Finger
{
    T left[4];
    T right[4];
    Finger<Node<T> > spine;
    int lcount;
    int rcount;
}
```

This successfully describes how a spine should operate, however there are two problems which could cause both a compile time and run time errors.

The problems lie within the line, `Finger<Node<T> > spine.` In C++, templates need to be resolved so that the compiler can determine all possible versions of the template which will be used and create them before their execution. This means that if a template class of int type is used and later in the same program one of a char type, the compiler will need to implement two versions of that class: one to accommodate ints and another chars. These as far as the program is concerned are two entirely different classes.

If a finger was created with an integer type, `Finger<int>,` the compiler would attempt to create a finger of type int. When it attempts to resolve the statement

`Finger<Node<T> >` it will be required to define a finger tree where the type will be a node of int. The compiler will attempt to define this and encounter the same statement repeatedly in the next finger tree and as it continues will attempt to resolve increasingly imbedded node types.

The method to resolve this is to have the compiler create a version of finger tree that will encompass all future levels and their type within the finger tree. The use of a void pointer, void*, will allow the storage of the address of an object which possesses the correct type for any level currently being operated on.

```
Finger<void*> spine;
```

The next issue is dealing with the declaration of the spine object. When a finger tree is declared at run time the program will keep creating the spine object within it, resulting in an endless loop. Stroustrup (1996, p. 867) indicates that if a pointer to a template is used instead then the object won't be instantiated at this point. By dynamically declaring finger tree's for the spine when they become necessary, the creation of this part of the structure can be avoided until it becomes a requirement. This also provides the advantage that in the absence of a spine object, where the spine pointer is null, it can easily be determined whether a tree is any deeper than the current level.

The correct definition is now shown below and at this stage it is possible to add some important boolean operations which will be in common use later.

```
template <typename T>
class Finger
{
    private:
    T left[4];
    T right[4];
    Finger<void*> *spine;
    int lcount;
    int rcount;

    bool single()const {return (!spine&&(lcount||rcount)&&!(lcount&&rcount));}
    bool ldangerous()const {return lcount==4;}
    bool rdangerous()const {return rcount==4;}
    public:
    Finger():lcount(0),rcount(0){}
    bool empty()const{return !(lcount||rcount||spine);}
}
```

3.2 Deque Operations

The next step is to provide methods to allow for the addition and removal of elements to a sequence. A deque, short for double ended queue, has operations which allow the pushing and popping of elements to either end of a list.

3.2.1 Pushing

Firstly, pushing items on to the back of the sequence will be examined. Hinze & Patterson (2006 p.5) defines four different states in which the finger tree can be found. These would need to be acted on differently when attempting to add to structure.

- If the tree is empty a *single* is created by adding to the right finger (back) of the tree.
- If the tree is a single then the currently stored item is moved to the left (front) and the new element is put onto the right (back), this will ensure the tree is sufficiently balanced if the operations are not random. It also allows the assumption that a tree level is always either a single or it will at all other times have an element on both the left and right fingers.
- If the right finger is full and contains four elements, three of them are removed and grouped together as a node of three elements and pushed on to the back of the spine. The new item can then be added to the back of the right finger behind the remaining item.
- If the Finger tree is not a single and the right hand side is not yet full then the element can be added to the back of the right.

Most of these processes can easily be defined in C++, the only part that requires additional thought is when the right is full. The spine will store a pointer to a node of type

T, albeit with a void pointer, so memory for this will need to be allocated through the *new* keyword. Once the node is created the elements from the right need to be copied in to it, the elements of this node on the spine will be closer to the front of the sequence than the items on the right of the current level. Therefore the first three elements on the right finger are taken.

The pointer, or address, of the node object is then pushed on to the back of the spine. If the spine is still null a new finger tree will need to be created and its address stored in the spine pointer. Here occurs the structures' first recursive call. As this process continues, the spine will have more nodes pushed onto it. Each level will need to perform the same checks and tasks whenever a node from the above level is pushed on to the back. When the right hand of the spine is full it will group three nodes together forming the 2-3 tree structures described earlier and push them onto the next level.

Pushing on to the front should follow the same process as push_back, although operations will be centred on the left finger, which represents the front of the stored sequence. When adding to the right finger in push_back it is easy to simply insert the new item in to the next unused part of the right array, when considering the structure as it has been presented so far then the left array could be interpreted as storing the front of the sequence at an index of 0. If the elements are stored in this manner the array would need to shuffle its currently stored elements down the array, which would be an unnecessarily expensive operation when compared to push_back. To avoid this reshuffling of the array as elements are inserted on to the front, the left finger can merely be reversed to act as the right array does, the higher the index in the left array the closer it is to the front. This concept can be visualised in figure 3.1.

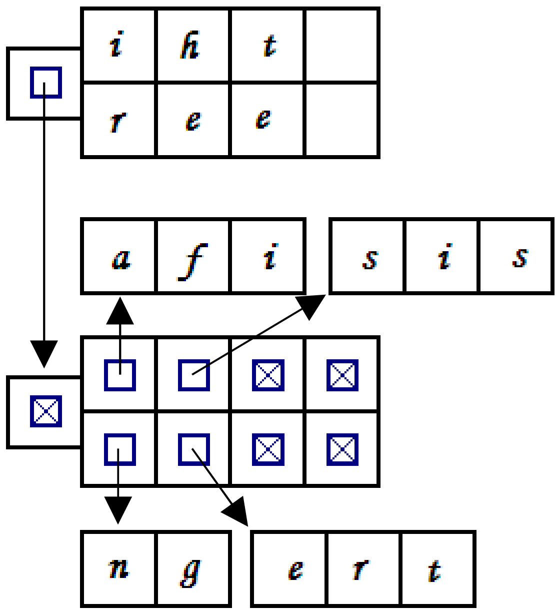


figure 3.1

As with push back, once left is full, a node will need to be created and pushed on to the level below. However elements in left have essentially been reversed, so that nodes will remain consistent across the structure and store elements in the expected sequence of left to right. Care must be taken to reverse the left fingers elements when they are added to a node so that this consistency occurs.

It has so far been established that the back of the finger tree will usually be the highest used index of the right finger (rcount-1) and the front will be the highest in the left (lcount-1). The only exception to this is when left or right are empty, but as established earlier this will only occur when the structure is a single, so this single element will be both the front and the back. With this information two very useful operations can be written i.e. those to retrieve the front and back elements of the sequence.

```
const T &front()const
{
    if (lcount)
        return (*left)[lcount-1];
    else // if (rcount)
        return (*right)[0];

};
```

```
const T &back()const
{
    if (rcount)
        return (*right)[rcount-1];
    else //if (lcount)
        return (*left)[0];
}
```

3.2.2 Popping

Next in the deque operations are popping operations to remove elements from the front and back of the finger tree. Popping elements from a finger tree is a simple procedure. Only items up to the left and right count will be accessed by reducing these counts the elements are essentially ignored and will be overwritten with later push operations. This applies when there are enough elements in the left and right. For all other cases care must be taken to enforce the presumption that the left and right will always contain at least one element each, otherwise the tree has no spine and is a single.

When there is only one remaining item in the left or right, the spine should be checked to establish if there are any available elements to be moved back up to the current level. The node on the spine containing the required elements can be accessed using the previously defined front and back operations, the node can then be popped off the spine using a recursive call. If the same situation is reached but there is no tree attached to the spine then an element corresponding to the correct end of the sequence will need to be moved from the left to the other side of the tree to enforce the assumption about singles. Finally if the tree is single, it becomes an empty and if the tree is empty no elements can be removed and no action is taken. The full code listing can be explored in Appendix B.

3.3 Persistence

There is an important aspect to programming this structure in C++ which so far has not been addressed. Memory management in C++ requires that each time memory is allocated it must be released with a delete operator. The finger tree definition so far is allocating memory when items are pushed on to the spine, as a result of this it would be sensible to deallocate this memory when popping items off the sequence. However the reason this has not been done is so that the structure can be made in to a persistent one.

Copying of the data structure is a common operation. The first logical way to achieve this is to merely copy each element one by one in to a new structure, however this is computationally expensive. By employing persistence through the structure it becomes possible to share stored elements.

This can be achieved with the current use of dynamic memory allocation and pointers. Copies of the structure made, need only obtain the memory addresses of the currently allocated memory to gain access to it. Hence the equals operator and copy constructor can be made straightforward.

```
Finger(const Finger &f)
{
     lcount=f.lcount;
     rcount=f.rcount;
     left=f.left;
     right=f.right;
     spine=f.spine;
}

Finger &operator=(const Finger &f)
{
     lcount=f.lcount;
     rcount=f.rcount;
     left=f.left;
     right=f.right;
     spine=f.spine;
     return *this;
}
```

At present the left and right arguments are defined as arrays of four elements, the compiler would not accept the initialisation of an array with a pointer to another one (stroustrup 1996, p.92). Therefore the solution would be to redefine these as pointers and allocate an array of four to them at initialisation. In this way the address can be successfully copied and the memory shared between finger tree data structures.

However there is much more to consider when sharing data like this. It must be ensured that changes to one of the finger trees do not affect any other copies. Consider the example where two finger trees share the left array, one copy requests to pop an item off the front, and consequently its lcount is reduced. At this point all will still function correctly, one tree has an lcount that will make use of all the elements stored in the array, the other will ignore the final element as it's lcount does not allow it to be accessed. However when the smaller finger attempts to push a new item on the front, the currently defined procedure will be instructed to overwrite whatever is contained in the array at its lcount, which the other finger tree is still making use of.

To resolve this, an independent count would need to be added to the left and right storage compartments. To avoid the overwriting of any elements that may be in use by another finger tree referencing this array of storage, the true number of elements that are stored or used within it must be known. Once this is ascertained this shard count can be compared to the lcount and rcount which is local to each finger tree to ensure items are not overwritten in the array when pushing items on to the structure. If the counts don't match then a copy of the affected part of the structure needs to be made so that it becomes independent of any other structures that were accessing it.

For this the arrays are encompassed into a class, similar to the node class used in the tree structures on lower levels but with an additional counter. Pointers to objects of this class replace left and right arrays and checks are added in to the currently defined deque operations to ensure persistence. For pushing a check will need to be added which will ascertain whether the referenced object, contains more items than expected.

For both popping and pushing any interaction with the spine requires a copy to be made. This is because the lcount and rcount of the spine will be affected and as other finger trees may have a pointer to the same spine, changes to the lcount and rcount would affect these too. One must remember however, the copy constructor for a finger tree merely involves copying of the data addresses and creating an independent lcount and rcount.

The procedure is simple and the new spine will still reference the same data as the other copies of the spine.

The result of sharing this memory however means that it becomes more difficult to keep track of how many finger trees are making use of memory at any one time. No single structure can take on the responsibility of deallocating memory as it will not have information about any other structures that are referencing it. The solution is to somehow keep a count of how many pointers to a piece of memory are in use and when no pointers are left, free the memory. This is an already well established process known as reference counting.

3.4 Reference Counting

When a space in memory is created for a data item or instantiation of a class as an object, any number of pointers can be created to this area of memory. Along with the created memory, reference counting stores a counter which tracks the number of pointers currently making use of this memory. When no pointers are addressing the memory it can be assumed that the address is no longer stored anywhere and it will not be accessible, hence it should be destroyed and the memory made available to the system again.

Reference counting can be implemented using two interacting classes. One class stores the record and a count of the references to it while the other acts as the accessor, in essence, the pointer to the record. The pointer class updates the records counter, incrementing it as a pointer is assigned to a record and decrementing it when either the pointer is destroyed or assigned to another record. The pointer will also destroy the record when the count reaches zero.

```
template <typename T>
class _ref_rec
{
      friend class ref_ptr<T>;

      int _ref_count;
      T _ref_value;
```

```
	_ref_rec():_ref_count(1){}
	_ref_rec(const T & arg) : _ref_value(arg),_ref_count(1){}
};

template <typename T>
class ref_ptr {

	_ref_rec<T> *_ptr;
public:
	ref_ptr() : _ptr(new _ref_rec<T>()) {}
	ref_ptr(const T & arg) : _ptr(new _ref_rec<T>(arg)) {}
	ref_ptr(const ref_ptr<T> & other) : _ptr(other._ptr) {
		_ptr->_ref_count++;
	}
	~ref_ptr() {
		if (--_ptr->_ref_count == 0)
			delete _ptr;
	}

	ref_ptr<T> & operator=(const ref_ptr<T> & other) {
		other._ptr->_ref_count++;     /* must be first! */
		if (--_ptr->_ref_count == 0)
			delete _ptr;
		_ptr = other._ptr;
		return *this;
	}

	T & operator*() const { return _ptr->_ref_value; }
	T * operator->() const { return &(_ptr->_ref_value); }
};
```

The above code demonstrates reference counting. The _ref_rec class holds the count and the instantiated data but is entirely private, ref_ptr is declared a friend of the ref_rec class so that it may have access to its data. The ref_ptr class holds the address of a ref_rec and at construction creates one unless it is created from another pointer in which case it takes the address of the ref_rec and increments its count. The destructor decrements the count and checks whether it is zero and if so deallocates the memory used by the ref_rec. A similar process takes place in the assignment operator, but first the count of the record which it is assigned to is incremented. This needs to be done first in case the pointer is assigned to the same ref_rec instance. If this hasn't been done first the record could be inadvertently deleted before the pointer can be reassigned to it. Overloading of * and -> allow for an object of ref_ptr to be used in the same way as a normal pointer.

These pointer types now need to be used in the finger tree structure.

```
ref_ptr<Node<T> > left;
ref_ptr<Node<T> > right;
```

Applying these to the left and right data items is simple, but problems occur when this implementation of reference counting are applied to the spine structure.

```
ref_ptr<Finger<ref_ptr<Node<T> > > spine;
```

The difficulties with defining templates for the spine which were discussed earlier are encountered again here. A ref_ptr by default will attempt to create the required data item within _ref_rec at its creation, attempting to make a finger tree in this case. It must not do this until it is absolutely necessary to avoid endless attempts to initialise the spine of each finger tree and so that, whether or not a spine pointer is null can be used as a recursive stopping condition. Hence a ref_ptr type which does not create an instantiation of its type through a _ref_rec until it is instructed to do so is required, it also should provide a boolean operator to advise whether or not it is null.

The other issue as before is the resolution of the type of the spine finger tree. Earlier this was resolved by defining a type which would cover all future spine types with a void pointer. So a reference pointer which has no concern over the type it references is needed, one that can hold any type without the specification of a template parameter.

This is difficult to resolve because at present both the record and the pointer class require a type, a records type is obtained from the type of pointer class and the stored object cannot be defined without a type to instantiate. However C++ does provide some facilities that may aid with this problem. C++ is able to deduce the type of an argument independently, without the need to specify the type name in the function call. This is known as template argument deduction. /reference/

```
template <typename T>
void_ref_ptr(const T & arg) : _ptr(new _ref_rec<T>(arg)) {}
```

A class which does not require a template argument can be achieved by removing the template type argument from the class and instead applying it to the functions within the class. In the example above the type T which is required by _ref_rec can be automatically determined by the argument to the constructor, so similarly to a null_ref_ptr a void_ref_ptr can only be instantiated properly by passing a data item of the required type to it. Any other functions within the class which require knowledge of the type stored will also need to determine it locally.

Other problems arise with the storage of the _ref_rec through a pointer and also with the calling of the destructor for it. In other pointer classes seen so far the only data stored is a pointer to a _ref_rec which is accessed and updated through these pointer classes. But this always requires the definition of the ref_rec type, which for this new class is unknown, except within the scope of the constructor and some other functions.

The first logical solution is to use a void pointer to accommodate the ref_rec. If any called function within the class requested a type, the void* could be cast to the appropriate ref_rec type so that access to it would be possible. However C++ will not allow for a destructor to be given a template argument. This is logical as the destructor is never invoked directly by a command from the programmer but rather through the automatic destruction of an object when it goes out of scope, or a request to free some previously allocated memory through the delete or delete[] commands. As the destructor for this pointer class requires access to the ref_recs' data, specifically its count but also indirectly it destructor, a void* will not suffice.

One possible solution becomes apparent when considering the requirements so far. A pointer to a ref_rec without a template argument is required, however, this pointer still needs to supply access to some of the elements of the ref_rec without the need for casting of the pointer. C++ inheritance relationships provide a solution. A ref_rec base class without a template type argument but with the data we require of it is defined. A ref_rec, which will hold the data of the required type, inherits from this. C++ allows for a pointer of a base type to hold the address of a derived type, without any *slicing* taking place.

Another consideration for this is to ensure that the correct destructor for the ref_rec is called, again C++ accommodates for this when a pointer to a class instantiation is used. By defining the destructor in the base class as virtual, the language will call the correct destructor for the object the pointer addresses whether or not it is a derived class.

Other issues are also encountered with the operator overloading of the pointer access methods * and ->. It is not possible to implement these operators with template deduction, as they accept no arguments that would include the template type. It is however possible to explicitly specify the template type for these operations when calling them.

However to call these defined functions with the template argument would require the writing of the full function name.

```
t.operator*<int>();
```

This makes the overloading of these operators irrelevant so it was decided to give the functions more appropriate names.

```
class _base_ref_rec {

      friend class void_ref_ptr;
      protected:
      int _ref_count;
      _base_ref_rec() : _ref_count(1) {}
      virtual ~_base_ref_rec(){}
};

template <typename T>
class _ref_rec : public _base_ref_rec
{
      friend class ref_ptr<T>;
      friend class null_ref_ptr<T>;
      friend class void_ref_ptr;
      T _ref_value;
      _ref_rec():_base_ref_rec(){}
      _ref_rec(const T & arg) : _base_ref_rec(),_ref_value(arg){}
}
```

```
class void_ref_ptr
{
    _base_ref_rec *_ptr;
public:
      void_ref_ptr() : _ptr(new _ref_rec<int>()) {}
      template <typename T>
      void_ref_ptr(const T & arg) : _ptr(new _ref_rec<T>(arg)) {}
      void_ref_ptr(const void_ref_ptr & other) : _ptr(other._ptr) {
            _ptr->_ref_count++;
      }


      ~void_ref_ptr() {
            if (--_ptr->_ref_count == 0)
                  delete _ptr;
      }

      void_ref_ptr & operator=(const void_ref_ptr & other);

      template <typename T>
      T & get() const { return ((_ref_rec<T>*)_ptr)->_ref_value; }
      template <typename T>
      T *get_ptr()const{return &(((_ref_rec<T>*)_ptr)->_ref_value);}
};
```

3.5 Test Environment

At this point of the implementation, testing of the current operations is required so that any errors in the code can be established. The testing would be required to remove run time errors and also to establish whether the finger tree was storing data in the correct manner.

For this initial test environment it was decided that operations should take place in a random sequence so that a variety of tree configurations could be achieved and as a result all areas of the code would be tested under a variety of conditions. The C++ STL includes a random number generator, rand(), with which a program can be made to execute a random sequence of operations. Using a switch statement to act on the modulus of a randomly generated integer can provide a suitable environment to test all aspects of the finger tree class. The rand() function does not generate a truly random sequence however, but instead relies on a seed value, set by the function srand(), to generate a sequence of numbers. If the seed value is the same each time the program is run then the generated sequence of random numbers will also always be the same. This actually provides a benefit to the test program as when an error occurs it can be reached again in future executions. It will also allow for the analysis of the operations that led up to the error, which could be at fault as appose the operation called at the point of program failure. The randomly generated numbers can also provide a source of data for the finger tree.

In order to examine what is held within the structure, a method to print out its contents is also required. The first implementation of this uses of the front() operation to obtain and print an element which can then be popped off the sequence. However it is soon observed that to truly examine what the state of the structure is, a print method which demonstrates what level data items are stored at and the structure of the nodes at each level, is required. This would need a method to access array elements of the left and right data items and display these no matter what type they may be including those which contain node trees Therefore moving down through the levels of these trees also needs to be considered. Two printing methods are required for this, one to handle the printing of normal data items and another for node trees.

As with the reference counting classes the use of inheritance is utilised to implement this print method. An abstract class is defined which provides a pure virtual method *print*. This accepts an ostream argument to send its output to along with the item which needs to be printed of template type T. Two classes are derived from this, *BasePrinter* which would print the type the tree is storing and a *NodePrinter* to handle node trees. Both of these need to implement the pure virtual method print. The BasePrinter simply outputs the passed element out to the accompanying ostream, consequently the passed type must have an appropriate overloading of the << operator.

For the NodePrinter to output its argument to the stream it would need to move through its stored elements and output these appropriately, but these could be just further nodes. So a variable is added to the NodePrinter of type printer and this type would then be established by a constructor which requests a printer argument. With this, the implemented print method outputs its arguments through this stored printer, either a NodePrinter or a Base printer if the bottom level of the node tree has been reached.

With these classes the finger tree print method which requests a printer type can output its left finger using the supplied printer argument, i.e. the top level would use a BasePrinter. It then calls the same function from the spine with a NodePrinter instead, using the current levels printer as the constructor argument for the NodePrinter. As this recursively moves down the spine, the stored printer within the NodePrinter will progressively contain embedded printers, matching the current level of the tree so that the nodes can be 'unwound' recursively and printed. The code for this print method can be found in Appendix B.

With this printing method established a test environment can be successfully explored. In order to examine how each operation is performing, the time taken to perform the requested instructions during the testing of the structure needs to be established. The ability to monitor this could be programmed in to the test environment but it is far simpler to use a profiler to gather this information. A profiler can examine the number of calls to a function, the time spent there and the functions it calls. It organises this data so that it is easy to see where a program is spending most of its time and how a function can be refined.

3.6 Concatenation

The shape of a finger tree allows for the concatenation of two sequences to be performed at a relatively fast speed, although by no means as efficient as a list. Hinze & Patterson (2006, p.9) argue that concatenation will take place in Θ(log n) where n is the number of elements contained within the smallest tree.

Consider the concatenation of the top level of two trees. The left of the first tree will still remain at the front of the sequence and the right of the second tree will remain at the end of the sequence after concatenation. Therefore the top level of the resultant concatenated tree will be made up of these areas of the two trees. The middle of the sequence will be made of the remaining items, i.e. the end of the first tree and the beginning of the second. These will need to be inserted into the spine of the new tree. As these elements must move down the spine they will need to be grouped into nodes, if the next level on either tree contains more elements it may be necessary to continue moving these middle elements further down the spine. Consequently the recursive calls will need to accommodate this middle sequence until it can be dealt with appropriately.

This grouping of middle elements in to nodes for lower levels of the tree presents the first need for nodes that may contain less than three elements. The left and right fingers from either tree should contain at least one element each, unless one is a single. If they do contain only one item each for the middle of the sequence then they must form a node of two elements. There could be a variety of amounts of elements made up from the right of the first and left of the second trees, these need to be accommodated for while avoiding the creation of a node of one element.

Already presented is the occurrence of a node being created out of two elements, similarly three elements can be grouped into a single node. Other amounts can be accommodated for with the following rules.

- Group two or three elements in to a node
- Group four elements into two equal nodes
- For any greater than four turn the first 3 items into a node and reapply any of these conditions to the remaining items

The first conditions of two or three elements can be grouped together as a single statement. This statement groups the elements together if there are three or less of them, case where there is only one element should not reach this point of the code. Four or more elements can be handled with similar method. If there are four elements, take two away and turn them in to a node, then perform the same checks again, the remaining two nodes should be handled by the first clause. If there are more than four then take three instead and continue examine the remaining elements again.

The following table shows how the elements would be split up by this procedure, remembering that at first the combination of the right of the first tree and left of the second could make up to a maximum of eight elements. But the table also shows how this maximum amount could grow, as nodes from the next level are combined with those passed down from above.

Number of elements	Node Sizes				
2	2				
3	3				
4	2	2			
5	3	2			
6	3	3			
7	3	2	2		At present the maximum number of nodes passed down is 3, a maximum of 8 nodes at the next level means up to 11 elements may need to be combined.
8	3	3	2		
9	3	3	3		
10	3	3	2	2	Here 4 nodes could be passed down, resulting in a new maximum of 12 elements to combine at the next level.
11	3	3	3	2	
12	3	3	3	3	

The code below demonstrates again how this splitting process is achieved.

```
Node<void_ref_ptr> tarr(4);
int pos=0;
int totalsz= back_f.lcount+front_f.rcount+arr.getCount();
do     //max 4 times
{
       if (totalsz>4)
              nodesz=3;
       else if (totalsz<4)
              nodesz=totalsz;
       else //if (totalsz==4)
              nodesz=2;

       Node<T> tmp(nodesz);
       while (tmp.getCount()<nodesz)
       {
              if (pos<front_f.rcount)
              {
                     tmp.push_back((*front_f.right)[pos]);
                     ++pos;
              }
```

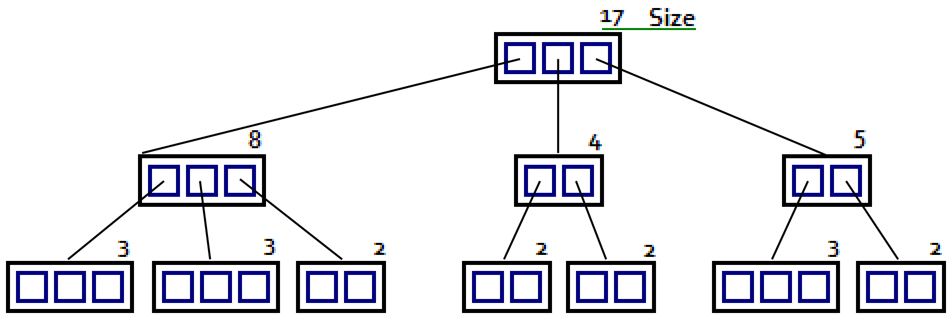
```
            else if (pos-front_f.rcount<arr.getCount())
            {
                  tmp.push_back(arr[pos-front_f.rcount]);
                  ++pos;
            }
            else //if (pos-front_f.rcount-arr.getCount()<back_f.lcount)
            {
                  tmp.push_back((*back_f.left)[back_f.lcount-1-(pos-
            front_f.rcount-arr.getCount())]);
                  ++pos;
            }
      }

      tarr.push_back(void_ref_ptr(tmp));
      totalsz-=nodesz;
} while (totalsz>0);
```

This recursive process of passing down the middle elements will stop when a level in which either tree is reached is either single or empty. If this is the case then the remaining items of the other tree becomes the rest of the new amalgamated tree and the nodes passed from the levels above can be pushed on to the back or front as appropriate, along with the single item in the other tree if there is one. This shows why the time complexity is dependant on the smallest tree as the remaining items in the longer tree require no further processing. The full code listing for concatenation can be found in appendix B.

3.7 Size

The ability of a data structure to maintain information about the number of elements it contains is fundamental to a useful data structure. While this could be implemented quite simply with the use of a single integer counter as items are added or removed from the structure, in a finger tree, information about size at various points across the structure will be an integral part of the implementation of further operations. These include access to any elements within the stored sequence and of the splitting of the structure around an element found in a similar manner. For these operations the structure will not only need information about its overall size but also the size of any node trees held along the spine of the structure.

Ideally the number of elements which are stored in a in a 2-3 tree could be stored within the tree itself as a single value. This way when a finger tree wishes to establish how many nodes are stored within a node type it could merely access the node to establish this. If the node tree is several levels deep it could establish how this total amount of elements are distributed across lower levels by moving down a level and obtain the total amount for each node tree stored within it. With this a finger tree can find any single element by comparing the index of it with the sizes stored and deducing which part of the tree should contain the element.



However when implementing this, difficulty is encountered with how to maintain these sizes as other operations such as push and pop take place, due to the recursive nature of the finger tree and its use of templates to accommodate its structure. At the top level, no node trees are stored, only elements of the required type. At lower levels nodes trees are stored. While it easy to ascertain that at the top level the true size of each element stored in the left and right compartments is one, lower levels would need to obtain their true size from the node stored within it. One finger tree layer would need to assume the size is one while the others would need to be instructed to obtain the size from the node. This difference between how the size is maintained cannot be expressed in a single template definition that covers all types.

To explain this better an example shall be explored, with push operations. A push operation is performed on a full left or right finger. Some of the currently stored items need to be put in to a node and the true size of this node will need to be established and set. For the top level it will be three elements so the true size is three. When this operation is performed at lower levels however the true size of the node will be the combination of the true sizes of the stored nodes.

The only common place that each level of the tree can access this size information is the left and right fingers of the tree. If a copy of the true size for the node trees is added to each storage elements and this is set to one for the top layer of a finger tree, then the size could be obtained from here at all levels. However this then becomes inadequate for pop operations.

A pop operation will need to move a node up from a lower level when there are insufficient elements at the current level. If the procedure described above is to be followed, the tree will need to then set the true sizes within the fingers so that future push operations have the information they require. However it only has the full size of the node and not information about how this size is distributed between the elements of the node which has been taken from the lower level. It would ideally copy this from the nodes stored within, but again at the top level these are just basic elements and the procedure will need to assume they have a value of one and deal with them differently from other levels. The resolution would require the nodes to store information about the true sizes of each of the elements they contain and these can be copied to the fingers.

This duplication of information about sizes, which should already be readily available in nodes, creates a lot of additional commands in the code for push and pop operations and will consequently make the code less efficient than it ought to be.

The implement concatenation procedure should be private to the class so that the only the class can have access to it. In order for template types for the top level to have access to private members of lower levels the following line needed to be added to the finger tree class.

```
template <class Y> friend class Finger<T>;
```

The line can be read as; any class of another type can have access to the private members of this class type.

4. Testing

The ultimate aim here is to establish whether a finger tree performs efficiently. To achieve this, a number of test environments are designed to fully examine each operation in a variety of different ways. All tests are performed on Intel Core2 Duo 3GHz processor.

4.1 Complete of All Operations

The first test environment would test all operations together and is similar to that used earlier to search for errors in the code. However no output to the console would be produced for this test as at this stage the code should be stable and no unnecessary calls should consume processor time while a function is being tested. Along with this user interaction is removed and the program is set to loop and perform large sets of random operations. These operations include push, pop, concatenation and also an additional operation to duplicate access to data through persistence, this occurrence could be included with the performance data. During the design of this environment however it was established that the addition of the concatenation procedure in to the tests produced extremely long sequences that would grow quite quickly. The aim of this test is to average operations across a wide variety of sequence sizes, so in order to avoid most iterations of the test dealing with a large sequence of operations, a maximum size value was incorporated into the procedure. To allow some leeway with this maximum size it was not merely tested against the size of the finger tree but integrated with the random numbers generated with the following condition.

```
random % max_size >= f1.size();
```

If this condition is met then the sequence should be sufficiently small enough to allow concatenation to take place. Otherwise the sequence would be reset to an empty tree so that it could grow again from a small value and proficiently test both large and small sequences. This max size condition was also added to push and pop operations so that the program would favour push operations when the sequence was small and pop when it was larger.

The condensed results obtained from gprof are shown below. Indented lines indicate operations which have been called by the function above them.

Function	Time in Function (s)	Time in Children (s)	Total time (s)	Calls	Single operation time (ns)
Finger<int>::operator+	0.1	81.24	81.34	10038661	810
Finger<int>::concat	1.38	79.64	81.02	10038661	807
Finger<void_ref_ptr>::concat	2.55	58.74	61.29	7820161	784
Finger<void_ref_ptr>::Finger	0.68	19.82	20.5	39255715	52
Finger<void_ref_ptr>::Finger	0.18	5.2	5.38	10298497	52
null_ref_ptr<Finger<void_ref_ptr> >()	0.13	3.93	4.06	15640322	26
Finger<int>::push_back	0.36	3.06	3.42	10016948	34
Finger<void_ref_ptr>::push_back	0.09	0.76	0.85	1842949	46
null_ref_ptr<Finger<void_ref_ptr> >()	0.02	0.46	0.48	1842949	26
Finger<int>::push_front	0.46	2.58	3.04	10030547	30
Finger<void_ref_ptr>::push_front	0.05	0.56	0.61	1855397	33
null_ref_ptr<Finger<void_ref_ptr> >()	0.02	0.47	0.49	1855397	26
Finger<int>::pop_back	0.06	0.93	0.99	2482476	40
Finger<void_ref_ptr>::pop_back	0.01	0.58	0.59	425178	139
null_ref_ptr<Finger<void_ref_ptr> >()	0	0.11	0.11	425178	26
Finger<int>::operator=	0.5	1.7	2.2	25000110	9
Finger<int>::~Finger	0.09	0.85	0.94	12497681	8
Finger<int>::pop_front	0.09	0.48	0.57	2469919	23
Finger<int>::Finger	0.02	0.27	0.29	2459020	12
Finger<int>::size	0.07	0	0.07	37497568	0

This first set of results provides a lot of information about how the implemented finger tree structure performs.

Concatenation shows a large performance time when compared to other operations. Little time is spent in the concat function for the top level. Instead most of the operational time is spent in its children and those of the lower levels. There is obviously significant overhead in performing the operations to form a new tree. This is demonstrated by the time spent creating new fingers. The reason for this seems to be the result of the creation of nodes in lower levels.

Push and pop operations seem to perform quite well at the point and the common operators such as size and the equals operators performance are not of concern.

4.2 Deque Operations with a Queue

In order to put the deque operations into perspective their operational times are next compared against those of an STL list. A suitable environment to examine these calls was created by using a queue to perform a large series of operations. A queue operates as a first in first out structure, abbreviated as FIFO. The first item to join a queue will be the first to be removed from it. It can also be considered as a last in, last out (LILO) as this conforms to the same rules that data is removed from the queue in the same order in which they are inserted. To implement a system like this, push operations can only take place on one end of the sequence and subsequent pop operations can only performed on the other end of the sequence.

For a fully comprehensive test of push and pop operations a large sequence is first prepared so that the structure can be sufficiently complex. For this a thousand elements are inserted into the structures before queue operations take place. Items are then pushed on to one end of the sequence and as this occurs an item is popped off the other end. To test both back and front operations the process is repeated but this time at opposite end of the sequence so that the queue is reversed. All operations which take place on a finger tree are duplicated with a list data structure so that the speed they both take to perform these operations can be compared.

```
for (int i=0;i<=10000000;i++)
{
    random=rand();
    f1.push_front(random);
    l1.push_front(random);
    f1.pop_back();
    l1.pop_back();
}
for (int i=0;i<=10000000;i++)
{
    random=rand();
    f1.push_back(random);
    l1.push_back(random);
    f1.pop_front();
    l1.pop_front();
}
```

So that the profiler can present useful performance time information the operations are repeated a number of times, ten million times each in this case. This extensive testing will also ensure stability within the structure.

Function	**Time in Function (s)**	**Time in Children (s)**	**Total time (s)**	**Calls**	**Single operation time (ns)**
Finger::push_front	0.63	4.91	5.54	10000491	55
std::list::push_front	0.11	0.6	0.71	10000491	7
Finger::push_back	0.32	5.06	5.38	10000509	54
std::list::push_back	0.08	0.61	0.69	10000509	7
Finger::pop_front	0.45	4.32	4.77	10000000	48
std::list::pop_front	0.06	0.27	0.33	10000000	3
Finger::pop_back	0.22	3.63	3.85	10000000	39
std::list::pop_back	0.05	0.24	0.29	10000000	3

As the table shows, the finger tree does not perform as well as a list, this was expected. Push operations can be seen here to take about eight times longer than a list, while pop operations take around fourteen times longer. This is an acceptable performance considering the structure of a finger tree when compared to a list. Push and pop operations on lists are not dependant on its size. For pushing, it merely has to allocate the space for the item and assign a pointer from the current list to this new memory. For popping the pointer is removed and the memory freed if necessary. A finger has to deal with the multiple layers for some of its operations in both push and pop. Looking at the full profile output it can be seen that a large proportion of the time of each operation is spent dealing with lower levels of the tree. The fact that these structures were loaded with a thousand items shows in the results. To store a thousand items would take up at least six layers within a finger tree, although not all operations will need to deal with these levels they will and to this programs operational time.

With this in mind it would be prudent to examine queue operations on a much shorter sequence, removing a finger trees need to operate on lower levels

Function	Time in Function (s)	Time in Children (s)	Total time (s)	Calls	Single operation time (ns)
Finger::push_front	0.06	0.33	0.39	10000002	4
std::list::push_front	0.04	0.67	0.71	10000002	7
Finger::push_back	0.17	0.39	0.56	10000002	6
std::list::push_back	0.12	0.65	0.77	10000002	8
Finger::pop_front	0.8	2.8	3.60	10000000	36
std::list::pop_front	0.05	0.6	0.65	10000000	7
Finger::pop_back	0.53	2.8	3.33	10000000	33
std::list::pop_back	0.16	0.57	0.73	10000000	7

This shows excellent results, with push operations actually outperforming those of a list. Pop operations however apply some reshuffling of elements from left to right or vice versa, in order to maintain the balance of the tree, so consequently these still perform worse than list. These results show that operations at each level are very efficient, as expected it is only the structure's interaction with lower level or its 'expensive' operations which add to its operational costs.

During the implementation it was discovered that the addition of the size operator would add significant overheads to push and pop operations. So it was decided that this test should be duplicated on a finger tree class with the additional commands to maintain size information removed.

Function	Time in Function (s)	Time in Children (s)	Total time (s)	Calls	Single operation time (ns)
Finger::push_front	0.57	5	5.57	10000487	56
std::list::push_front	0.14	0.5	0.64	10000487	6
Finger::push_back	0.32	4.95	5.27	10000513	53
std::list::push_back	0.09	0.49	0.58	10000513	6
Finger::pop_front	0.40	4.34	4.74	10000000	47
std::list::pop_front	0.09	0.37	0.46	10000000	5
Finger::pop_back	0.21	3.6	3.81	10000000	38
std::list::pop_back	0.07	0.34	0.41	10000000	4

These results show that the removal of size operations has little impact on the deque operation. The performance here is shows little improvement over previous results…

4.3 Concatenation

Here the concatenation operator is compared to the concatenation procedure for a list. To perform concatenation with a list the insert method must be used to insert elements from one list on to the end of another one.

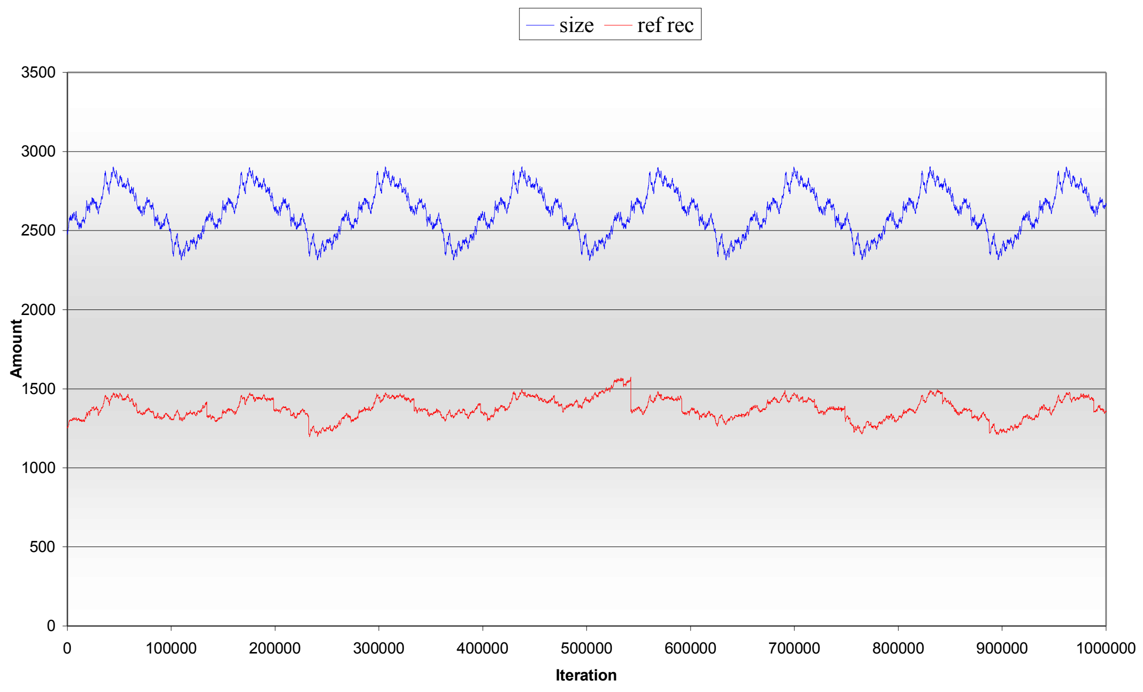
Function	Time in Function (s)	Time in Children (s)	Total time (s)	Calls	Single operation time (ns)
Finger<int>::operator+	0.03	3.81	3.84	300004	1280
std::list<int>::insert	0	1.38	1.38	300004	460

The above table shows the operational times of the concatenation of list and finger trees, across a wide spectrum of sizes. The finger trees' implementation performs at about two and a half times slower than that of a list. An STL list concatenation through the use of the insert method operates in linear time, due to its initialisation of elements as they are added to the sequence. Consequently the anticipation of the concatenation procedure to operate O(log n) complexity is not achieved here.

4.4 Memory management

Testing whether all memory is reclaimed can be achieved with a relatively simple program. Static count variables are added to the reference classes which are instructed to increment these counts at their construction and decrement only at destruction. These counts can then be used to indicate how many of the classes are in use and consequently, whether these values grow and shrink appropriately, thereby indicating if they are being destroyed and the memory made available as a result. When a large series of instructions are executed, the counts should remain at a reasonable amount for the data being stored. If the amount of objects continues to grow to an unexpected level this will be an indication of memory not being freed appropriately.

Either of the test environments defined so far would be appropriate for these tests by merely adding output to the console to display the number of reference counting classes are active. Both the interactive and queue tests were adapted for these experiments. The interactive program provides a good way to monitor whether the reference classes are behaving as expected at each command. The queue program performs, as before, a large amount of both push and pop operations. It was adapted slightly to vary the occurrence of these operations so that the sequence size could vary but despite this if successful the memory usage should remain within certain bounds.



The graph shows that over one million iterations, the use of memory remains steady as the size of the finger tree varies with push and pop operations. This demonstrates the implemented reference counting techniques were successful and memory resources are being freed as required.

5. Discussion

The results show that these operations perform relatively well, taking operational times that would be acceptable for this kind of data structure.

Push operations are shown to have an average operational time of 52 ns and pops 41 ns. These times are less favourable than those used in a list structure. However these longer times were anticipated from the outset because of the amortisation of the cheaper operations taking place at higher levels of the structure combined with the progressively expensive operations at lower levels.

Persistence of the structure was also successfully implemented, yet it can be seen that including this has significantly added to each of the operations performance. The need to create a new spine with every operation which affects it can be seen in the results to make the most impact on operational times. While copying of a tree is a very simple operation the regular needs for it to be performed adds heavily to performance. For example with continual push operations one in four will require this copying of the spine and the square of this amount of operations will effect the following layer and so on for deeper trees. Although this was anticipated but it was not expected to be as time consuming. Yet the results show that the creation of the reference counted record takes a significantly noticeable amount of time.

Persistence also required the use of reference counting in C++ in order to manage used storage and use memory resources responsibly. The results show that these were successful with memory being effectively released during operations. Yet the proportion of resources used for this also noticeably affects operational time. This can be traced to a significant time spent instantiating nodes used for the 2-3 node trees.

6. Conclusions

The finger tree structure suggested by Hinze & Patterson (2006) aims to provide a relatively simple, functional and persistent data structure which would provide good performance across a wide selection of common operations, rather than other alternatives which either performs well for some but not in all or can be overly complex to implement. This work has provided the first known attempt to implement such a data structure in C++. Although not all operations have been explored, the basic structure of a finger tree and the requirements to support these additional functions has successfully been implemented.

The implementation of the basic structure was approached in the most appropriate manner thought available in C++. With the use of templates this report shows that the tree structure could be successfully implemented and all the requirements of storage be met. However as more functionality was added to the structure it became increasingly difficult to apply what should have been simple processes and the limits of using templates became apparent. The exploration of templates here has explored the ways in which C++ deals with the generalising of data types, such as its requirement to resolve template classes for each newly defined type argument. With this work this meant that a work around had to be found so that all levels of a tree could be accommodated for by the code used within a single template class. The workaround explored which used void pointers is common therefore this was relatively simple to resolve.

Another area that needed to be addressed was that C++ has no built in memory management or 'garbage collection' routines. Instead, memory management is left up to the developer to implement effectively, which many people have done successfully, often to the benefit of an application which compared to a catch all memory management system, like that of Java, would be insufficient.

The most commonly used resolution to this problem is reference counting which, in a way, allows a program to manage its memory use itself. Unfortunately when reference counting was added to the finger tree, templates again added further complexities to the problem. The previous issue of template class resolution meant that reference counting had to be implemented without template arguments, yet still be able to store and manage multiple argument types. This, once more, had to be accommodated for with the facilities available in C++, which ultimately resulted in the less than ideal solution seen in void_ref_ptr.

Later adding size information was also made more complicated because of the restrictions of templates. Interaction between classes of different template types seems limited from the findings in this work. It would have been beneficial if it were possible to add conditions based on the template type and added to code within a class to allow a subset of statements to be executed if the template type matched such a condition.

Alternative solutions which could be explored to resolve some of these issues caused by templates would result in code that was less compact than that produced here. These could include the writing of two separate finger tree classes, one to accommodate the top level of a finger tree and another to handle lower levels. This would require duplication of all the functions within the classes even if they were similar, although it may be possible to use inheritance as a work around for this.

With the addition of these problems some sections of the produced class contained additional statements and sub-functions that ultimately led to poorer performance of the overall structure and these were demonstrated in the test results.

Further work will include the implementation of a method for random access and with this a way of splitting a sequence. The ground work for these methods is already available from the work done to add size information to the structure. Also the efficiency of push and pop operations could be improved by delaying operations which take place on the spine. In the implementation shown here these operations are done immediately, which results in need to duplicate the spine at every operation on it and as shown in the results this adds significantly to performance times of deque operations.

While completing this work much has been learnt about data structures and their uses as well as how to measure performance of code and how to act on those results accordingly. The exploration of many aspects of C++ has allowed the gaining of knowledge regarding techniques of using them and their limitations. Also problems which had not been previously explored and the examination of the known solutions to these, will hopefully allow for the adaptation of the findings here when pursuing other problems.

References

Heileman, G 1996, Data Structures, Algorithms, and object-orientated programming, McGraw-Hill

Hinze, R and Paterson, R 2006, Journal of Functional Programming pp 197-217, Retrieved: June 20, 2008, from http://www.soi.city.ac.uk/~ross/papers/FingerTree.html

Horowitz, E, Sahni, S, Rajasekaran, S 1997,Computer Algorithms C++, W. H. Freeman and Company

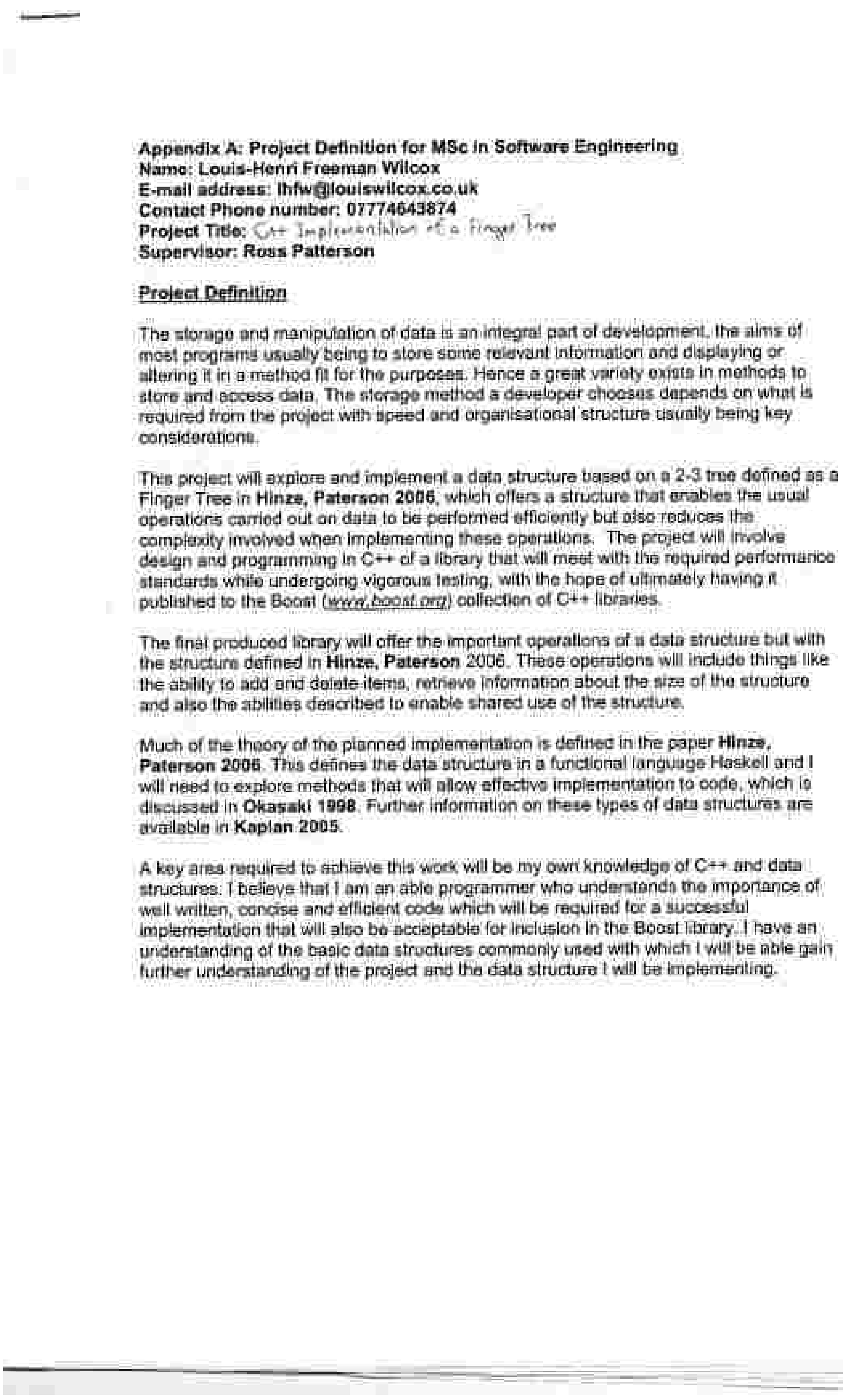
Kaplan, H 2005 Persistent data structures, In Handbook on Data Structures and Applications, D. Mehta and S. Sahni, editors, CRC Press, Retrieved: June 20, 2008, from http://www.math.tau.ac.il/~haimk/papers/persistent-survey.ps

Kingston, J 1998, Algorithms and Data Structures Design, Correctness, Analysis, 2nd edn, Addison Wesley Longman Ltd

Okasaki, C 1998, Purely Functional Data Structures, Cambridge, University Press, Retrieved: June 20, 2008, from http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf

Stroustrup, B 1997, The C++ Programming Language, 3rd edn, AT&T

Appendix A – Project Definition



Appendix A: Project Definition for MSc in Software Engineering
Name: Louis-Henri Freeman Wilcox
E-mail address: lhfw@louiswilcox.co.uk
Contact Phone number: 07774643874
Project Title: C++ Implementation of a Finger Tree
Supervisor: Ross Patterson

Project Definition

The storage and manipulation of data is an integral part of development, the aims of most programs usually being to store some relevant information and displaying or altering it in a method fit for the purposes. Hence a great variety exists in methods to store and access data. The storage method a developer chooses depends on what is required from the project with speed and organisational structure usually being key considerations.

This project will explore and implement a data structure based on a 2-3 tree defined as a Finger Tree in **Hinze, Paterson 2006**, which offers a structure that enables the usual operations carried out on data to be performed efficiently but also reduces the complexity involved when implementing these operations. The project will involve design and programming in C++ of a library that will meet with the required performance standards while undergoing vigorous testing, with the hope of ultimately having it published to the Boost (*www.boost.org*) collection of C++ libraries.

The final produced library will offer the important operations of a data structure but with the structure defined in **Hinze, Paterson** 2006. These operations will include things like the ability to add and delete items, retrieve information about the size of the structure and also the abilities described to enable shared use of the structure.

Much of the theory of the planned implementation is defined in the paper **Hinze, Paterson 2006**. This defines the data structure in a functional language Haskell and I will need to explore methods that will allow effective implementation to code, which is discussed in **Okasaki 1998**. Further information on these types of data structures are available in **Kaplan 2005**.

A key area required to achieve this work will be my own knowledge of C++ and data structures. I believe that I am an able programmer who understands the importance of well written, concise and efficient code which will be required for a successful implementation that will also be acceptable for inclusion in the Boost library. I have an understanding of the basic data structures commonly used with which I will be able gain further understanding of the project and the data structure I will be implementing.

Project Plan

The development of the project can be defined in three separate stages below some of these areas will overlap during the course of the project however they should usually progress from one to next. Key deliverables of the final code will be a stable and flexible library, which operates efficiently. These will be tested using the methods below and also compared with current implementations of similar data structures.

Literature Review and Research

This will not only need to take place at the beginning of the project but also during early implementation as more understanding of the project will be gained through initial builds.

Implementation and Testing

This will cover the majority of the time allocated for the project and will take place in progressive steps towards a final implementation. Testing of the implementation will be done through out the development, to ensure the code is successfully executing without errors. Also, suitable test data will need to be established in order ensure the code performs as expected and in an efficient manner. Some of the envisaged stages are defined below.

- Basic data structure with deque operations
- Design and implement a suitable test harness
- Add reference counting and consequently ensure efficient memory management
- Benchmarking
- Implement append operation
- Add sizes and implement position-based operations
- Implement delaying of deque operations on subtrees to allow structures to be shared

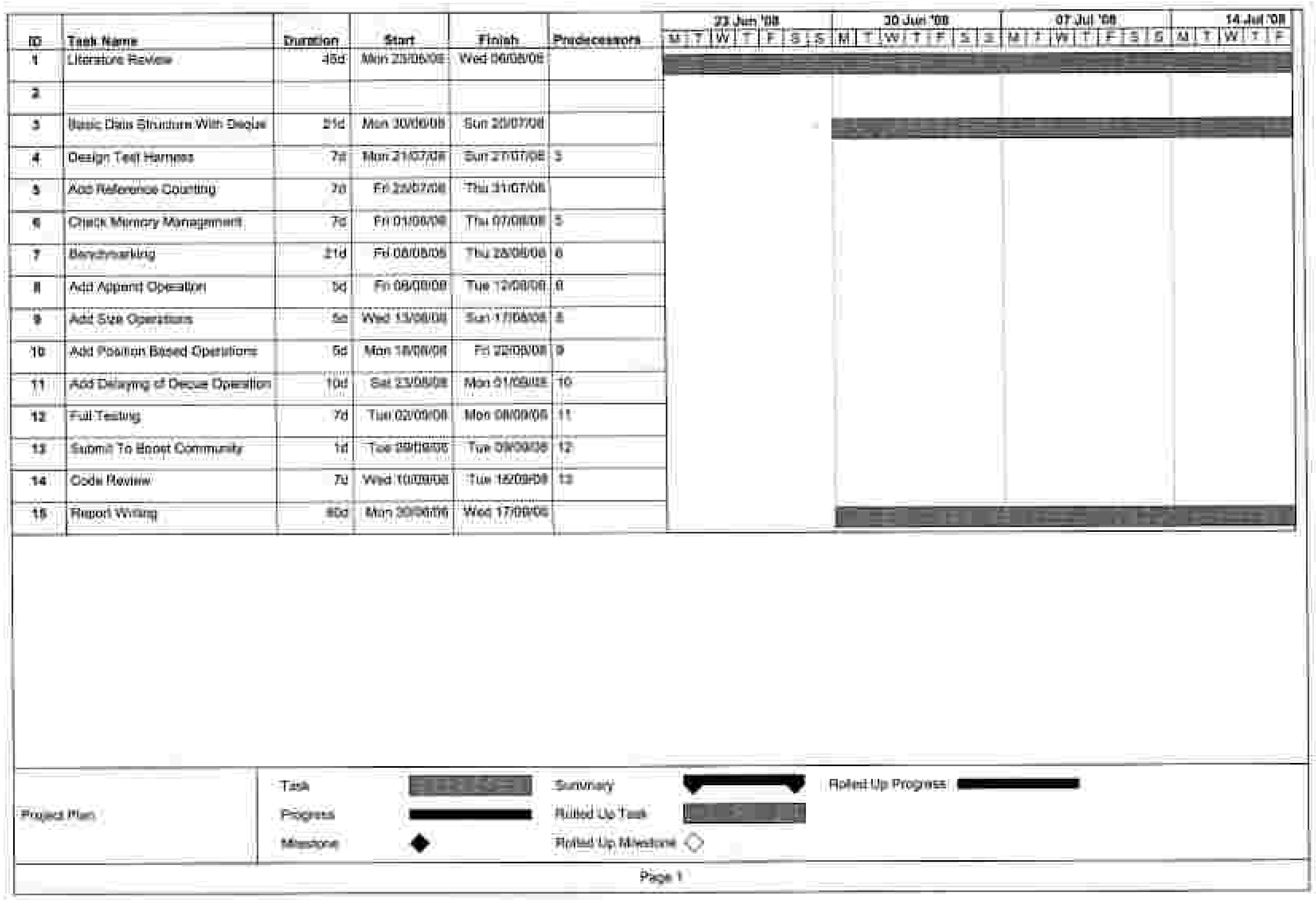
Publication

Submission to the Boost community will enable me to establish implementation reviews required following community feedback for publication.

References

1. Finger Trees: A Simple General-purpose Data Structure, by Ralf Hinze and Ross Paterson vol. 16(2), pp197-217, 2006. http://www.soi.city.ac.uk/~ross/papers/FingerTree.html
2. Purely Functional Data Structures, by Chris Okasaki. Cambridge University Press, 1998 http://www.cs.cmu.edu/~rwh/theses/okasaki.pdf
3. Persistent data structures, by H. Kaplan, In Handbook on Data Structures and Applications, D. Mehta and S. Sahni, editors, CRC Press, 2005 http://www.math.tau.ac.il/~haimk/papers/persistent-survey.ps

[illegible]

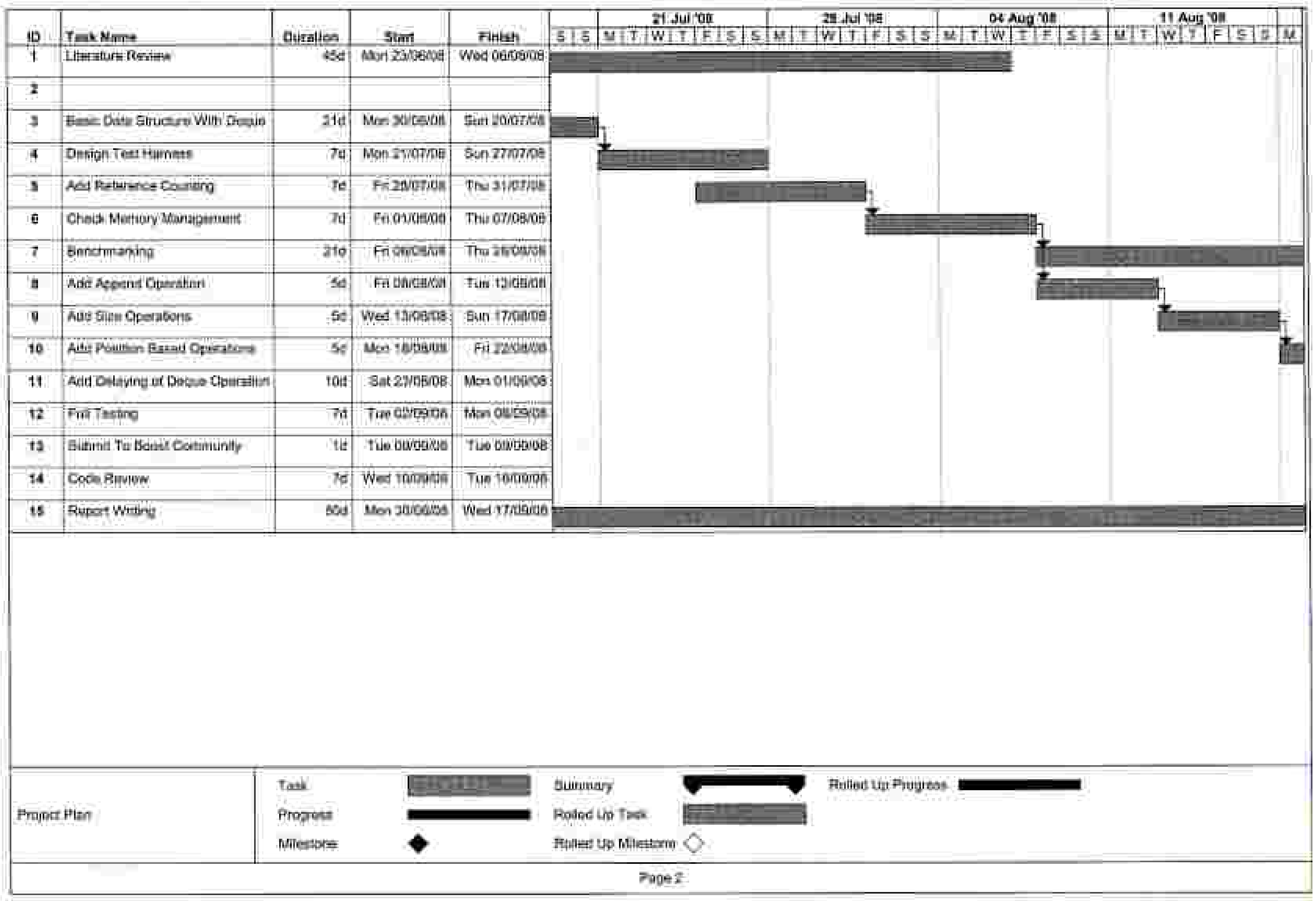


ID	Task Name	Duration	Start	Finish	Predecessors
1	Literature Review	45d	Mon 23/06/08	Wed 06/08/08	
2					
3	Basic Data Structure With Deque	21d	Mon 30/06/08	Sun 20/07/08	
4	Design Test Harness	7d	Mon 21/07/08	Sun 27/07/08	3
5	Add Reference Counting	7d	Fri 25/07/08	Thu 31/07/08	
6	Check Memory Management	7d	Fri 01/08/08	Thu 07/08/08	5
7	Benchmarking	21d	Fri 08/08/08	Thu 28/08/08	6
8	Add Append Operation	5d	Fri 08/08/08	Tue 12/08/08	6
9	Add Size Operations	5d	Wed 13/08/08	Sun 17/08/08	8
10	Add Position Based Operations	5d	Mon 18/08/08	Fri 22/08/08	9
11	Add Delaying of Deque Operation	10d	Sat 23/08/08	Mon 01/09/08	10
12	Full Testing	7d	Tue 02/09/08	Mon 08/09/08	11
13	Submit To Boost Community	1d	Tue 09/09/08	Tue 09/09/08	12
14	Code Review	7d	Wed 10/09/08	Tue 16/09/08	13
15	Report Writing	80d	Mon 30/06/08	Wed 17/09/08	

Project Plan

Page 1

Appendix A – Project Definition

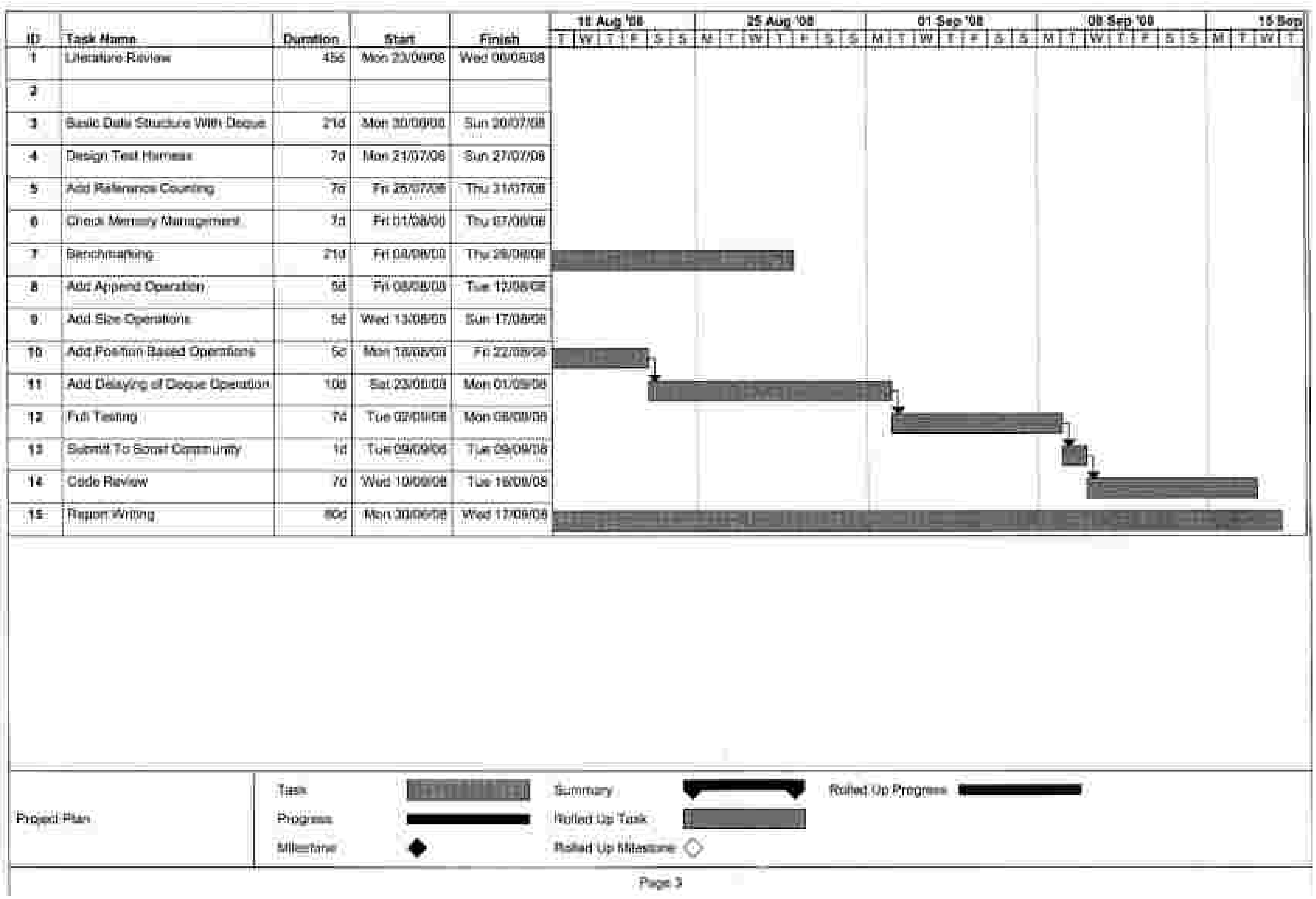


ID	Task Name	Duration	Start	Finish
1	Literature Review	45d	Mon 23/06/08	Wed 06/08/08
2				
3	Basic Data Structure With Deque	21d	Mon 30/06/08	Sun 20/07/08
4	Design Test Harness	7d	Mon 21/07/08	Sun 27/07/08
5	Add Reference Counting	7d	Fri 25/07/08	Thu 31/07/08
6	Check Memory Management	7d	Fri 01/08/08	Thu 07/08/08
7	Benchmarking	21d	Fri 08/08/08	Thu 28/08/08
8	Add Append Operation	5d	Fri 08/08/08	Tue 12/08/08
9	Add Size Operations	5d	Wed 13/08/08	Sun 17/08/08
10	Add Position Based Operations	5d	Mon 18/08/08	Fri 22/08/08
11	Add Delaying of Deque Operation	10d	Sat 23/08/08	Mon 01/09/08
12	Full Testing	7d	Tue 02/09/08	Mon 08/09/08
13	Submit To Boost Community	1d	Tue 09/09/08	Tue 09/09/08
14	Code Review	7d	Wed 10/09/08	Tue 16/09/08
15	Report Writing	80d	Mon 30/06/08	Wed 17/09/08

Project Plan						
	Task	▨	Summary	▬	Rolled Up Progress	▬
	Progress	▬	Rolled Up Task	▨		
	Milestone	◆	Rolled Up Milestone	◇		

Page 2

Appendix A – Project Definition



ID	Task Name	Duration	Start	Finish
1	Literature Review	45d	Mon 23/06/08	Wed 06/08/08
2				
3	Basic Data Structure With Deque	21d	Mon 30/06/08	Sun 20/07/08
4	Design Test Harness	7d	Mon 21/07/08	Sun 27/07/08
5	Add Reference Counting	7d	Fri 25/07/08	Thu 31/07/08
6	Check Memory Management	7d	Fri 01/08/08	Thu 07/08/08
7	Benchmarking	21d	Fri 08/08/08	Thu 28/08/08
8	Add Append Operation	5d	Fri 08/08/08	Tue 12/08/08
9	Add Size Operations	5d	Wed 13/08/08	Sun 17/08/08
10	Add Position Based Operations	5d	Mon 18/08/08	Fri 22/08/08
11	Add Delaying of Deque Operation	10d	Sat 23/08/08	Mon 01/09/08
12	Full Testing	7d	Tue 02/09/08	Mon 08/09/08
13	Submit To Boost Community	1d	Tue 09/09/08	Tue 09/09/08
14	Code Review	7d	Wed 10/09/08	Tue 16/09/08
15	Report Writing	80d	Mon 30/06/08	Wed 17/09/08

18 Aug '08
25 Aug '08
01 Sep '08
08 Sep '08
15 Sep
T W T F S S M T W T F S S M T W T F S S M T W T F S S M T W T

Project Plan

Task	▨	Summary	▬	Rolled Up Progress	▬
Progress	▬	Rolled Up Task	▨		
Milestone	◆	Rolled Up Milestone	◇		

Page 3

Appendix B – Code Listings

Finger.h

```
#ifndef _Finger_h
#define _Finger_h

#include <iostream>
#include "Node.h"
#include "refptr.h"

using namespace std;

template <typename T>
class Printer
{
public:
      virtual ostream & print(ostream & out, const T & x) const = 0;
};

template <typename T>
class BasePrinter : public Printer<T>
{
public:
      ostream & print(ostream & out, const T & x) const
{
            return out << x;
      }
};

template <typename T>
class NodePrinter : public Printer<void_ref_ptr> {
      const Printer<T> & elem_printer;
public:
      NodePrinter(const Printer<T> & p) : elem_printer(p) {}

      ostream & print(ostream & out, const void_ref_ptr & x) const {
            Node<T> p;
            p=x.get<Node<T> >();
            out << '(';
            int n = p.getCount();
            for (int i = 0; i < n; i++) {
                  if (i > 0)
                        out << ", ";
                  elem_printer.print(out, p[i]);
            }
            out << ')';
            return out;
      }
};

template <typename T>
class Finger
{
    private:
    template <class Y> friend class Finger<T>;

    ref_ptr<Node<T> > left;
    ref_ptr<Node<T> > right;
    null_ref_ptr<Finger<void_ref_ptr > > spine;
    unsigned int lcount;
    unsigned int rcount;
    unsigned int true_size;

    bool single()const {return (spine.isNull()&(lcount||rcount)&&!
(lcount&&rcount));}
      bool deep(){return (!spine.isNull());};
      bool ldangerous()const{return lcount==4;};
      bool rdangerous()const{return rcount==4;};
      const Finger concat(const Finger &front_f, const Finger
&back_f,Node<T> &arr)const;
    ostream & print_aux(ostream & out, const Printer<T> & pr) const;

    public:

    Finger():left(Node<T>(4)),right(Node<T>(4)),lcount(0),rcount(0),tru
e_size(0){}
    Finger(const Finger &f);
    Finger &operator=(const Finger &f);

      bool empty()const{return lcount==0&&rcount==0&&spine.isNull();};
    unsigned int size()const {return true_size;}

    void push_back(const T &val);
    void push_front(const T &val);
    void pop_back();
    void pop_front();

    const Finger operator+(const Finger &f)const
    {
        Node<T> u;
        return concat(*this,f,u);
    }

    const T &front()const
    {
        if (lcount)
            return (*left)[lcount-1];
        else // if (rcount)
            return (*right)[0];

    };

    const T &back()const
    {
        if (rcount)
            return (*right)[rcount-1];
        else
            return (*left)[0];
    }

      ostream & print(ostream & out) const
      {
            return print_aux(out, BasePrinter<T>());
      }
```

```
};


template <typename T>
Finger<T>::Finger(const Finger &f)
{
    lcount=f.lcount;
    rcount=f.rcount;
    left=f.left;
    right=f.right;
    spine=f.spine;
    true_size=f.true_size;
};

template <typename T>
Finger<T> &Finger<T>::operator=(const Finger &f)
{
    lcount=f.lcount;
    rcount=f.rcount;
    left=f.left;
    right=f.right;
    spine=f.spine;
    true_size=f.true_size;
    return *this;
};

template <typename T>
void Finger<T>::push_back(const T &val)
{
    if (empty())
    {
        if(right->getCount())
            right=ref_ptr<Node<T> >(Node<T>(4));
        right->push_back(val);
        ++rcount;
    }
    else if (single())
    {
        if (rcount)
        {
            if(left->getCount())
                left=ref_ptr<Node<T> >(Node<T>(4));
            left->push_back((*right)[0]);
            left->set_true_size(0,right->get_true_size(0));
            right=ref_ptr<Node<T> >(Node<T>(4));
            right->push_back(val);
            ++lcount;
        }
        else if (lcount)
        {
            if(right->getCount())
                right=ref_ptr<Node<T> >(Node<T>(4));
            right->push_back(val);
            ++rcount;
        }
    }
    else if (rdangerous())
    {
        if (!deep())
            spine=null_ref_ptr<Finger<void_ref_ptr >
>(Finger<void_ref_ptr >());
        else
            spine=null_ref_ptr<Finger<void_ref_ptr> >(*spine);
        Node<T> tnode(3);
        tnode.push_back(*right);//true_size copied from right

        unsigned int tnode_true_size=right->get_true_size(0)+right-
>get_true_size(0)+right->get_true_size(0);

        void_ref_ptr snode(tnode);
        spine->push_back(snode);
        spine->right->set_true_size(spine->rcount-1,tnode_true_size);
        spine->true_size+=tnode_true_size-1;
        ref_ptr<Node<T> > tmp(Node<T>(4));
        tmp->push_back((*right)[3]);
        tmp->set_true_size(0,right->get_true_size(3));
        tmp->push_back(val);
        right=tmp;
        rcount=2;
    }
    else
    {
        if(rcount<right->getCount())
        {
            ref_ptr<Node<T> > tnode=ref_ptr<Node<T> >(Node<T>(4));
            for(unsigned int i=0;i<rcount;++i)
                tnode->push_back((*right)[i]);
            right=tnode;
        }
        right->push_back(val);
        ++rcount;
    }
    ++true_size;
}

template <typename T>
void Finger<T>::push_front(const T &val)
{
    if (empty())
    {
        if(left->getCount())
            left=ref_ptr<Node<T> >(Node<T>(4));
        left->push_back(val);
        ++lcount;
    }
    else if (single())
    {
        if (lcount)
        {
            if(right->getCount())
                right=ref_ptr<Node<T> >(Node<T>(4));
            right->push_back((*left)[0]);
            right->set_true_size(0,left->get_true_size(0));
            left=ref_ptr<Node<T> >(Node<T>(4));
            left->push_back(val);
            ++rcount;
        }
        else if (rcount)
        {
            if(left->getCount())
                left=ref_ptr<Node<T> >(Node<T>(4));
            left->push_back(val);
            ++lcount;
        }
    }
    else if (ldangerous())
    {
        if (!deep())
            spine=null_ref_ptr<Finger<void_ref_ptr >
>(Finger<void_ref_ptr>());
        else
            spine=null_ref_ptr<Finger<void_ref_ptr > >(*spine);

        Node<T> tnode(3);
        unsigned int tnode_true_size=0;
        for (int i=2; i>=0;--i)
        {
            tnode.push_back((*left)[i]);
            tnode.set_true_size(i,left->get_true_size(2-i));
            tnode_true_size+=left->get_true_size(i);
        }
        void_ref_ptr snode(tnode);
        spine->push_front(snode);
        spine->left->set_true_size(spine->lcount-1,tnode_true_size);
        spine->true_size+=tnode_true_size-1;
        ref_ptr<Node<T> > tmp(Node<T>(4));
        tmp->push_back((*left)[3]);
        tmp->push_back(val);
        tmp->set_true_size(0,left->get_true_size(3));
        left=tmp;
        lcount=2;
    }
    else
    {
        if(lcount<left->getCount())
        {
            ref_ptr<Node<T> > tnode=ref_ptr<Node<T> >(Node<T>(4));
            for(unsigned int i=0;i<lcount;++i)
            {
                tnode->push_back((*left)[i]);
                tnode->set_true_size(i,left->get_true_size(i));
            }
            left=tnode;
        }
        left->push_back(val);
        ++lcount;
    }
    ++true_size;
}

template <typename T>
```

```
void Finger<T>::pop_back()
{
    if (rcount>1)
    {
        --rcount;
        true_size-=right->get_true_size(rcount);
    }
    else if (!spine.isNull())
    {
        true_size-=right->get_true_size(0);
        void_ref_ptr b(spine->back());
        right=ref_ptr<Node<T> >(Node<T>(b.get<Node<T> >(),4));
        rcount=right->getCount();

        spine=null_ref_ptr<Finger<void_ref_ptr> >(*spine);
        spine->pop_back();

        if (spine->empty())
                spine=null_ref_ptr<Finger<void_ref_ptr > >();//make
spine null
    }
    else if (lcount>1)
    {
            true_size-=right->get_true_size(0);
            right=ref_ptr<Node<T> >(Node<T>(4));
            right->push_back((*left)[0]);
            right->set_true_size(0,left->get_true_size(0));
            ref_ptr<Node<T> > tmp(Node<T>(4));
            for (unsigned int i=1;i<lcount;i++)
            {
                tmp->push_back((*left)[i]);
                tmp->set_true_size(i-1,left->get_true_size(i));
            }
            left=tmp;
            --lcount;
    }
    else if (lcount||rcount)
    {
        if (rcount)
        {
            --rcount;
            true_size-=right->get_true_size(rcount);
        }
        else
        {
            --lcount;
            true_size-=left->get_true_size(lcount);
        }
    }
    //else is empty
}

template <typename T>
void Finger<T>::pop_front()
{
    if (lcount>1)
    {
        --lcount;
        true_size-=left->get_true_size(lcount);
    }
    else if (!spine.isNull())
    {
        true_size-=left->get_true_size(0);
        left=ref_ptr<Node<T> >(Node<T>(4));
        void_ref_ptr f(spine->front());
        ref_ptr<Node<T> > tmp(f.get<Node<T> >());

        for (int i=tmp->getCount()-1;i>=0;--i)
        {
            left->push_back((*tmp)[i]);
            left->set_true_size(lcount,tmp->get_true_size(i));
        }
        lcount=left->getCount();

        spine=null_ref_ptr<Finger<void_ref_ptr> >(*spine);
        spine->pop_front();
        if (spine->empty())
            spine=null_ref_ptr<Finger<void_ref_ptr > >();
    }
    else if (rcount>1)
    {
        true_size-=left->get_true_size(0);
        left=ref_ptr<Node<T> >(Node<T>(4));
        left->push_back((*right)[0]);
        left->set_true_size(0,right->get_true_size(0));
        ref_ptr<Node<T> > tmp(Node<T>(4));
        for (unsigned int i=1;i<rcount;i++)
        {
            tmp->push_back((*right)[i]);
            tmp->set_true_size(i-1,right->get_true_size(i));
        }
        right=tmp;
        --rcount;
    }
    else if (lcount||rcount)
    {
        if (lcount)
        {
            --lcount;
            true_size-=left->get_true_size(lcount);
        }
        else
        {
            --rcount;
            true_size-=right->get_true_size(rcount);
        }
    }
    //else is empty
};


template <typename T>
```

```
const Finger<T> Finger<T>::concat(const Finger &front_f, const Finger
&back_f,Node<T> &arr)const
{
    Finger out;

    if (front_f.empty())
    {
        out=back_f;
        for (int i=arr.getCount()-1;i>=0;--i)
        {
            out.push_front(arr[i]);
            out.left->set_true_size(out.lcount-1,arr.get_true_size(i));
        }
        out.true_size=front_f.true_size+back_f.true_size;
        return out;
    }
    else if (back_f.empty())
    {
        out=front_f;
        for (unsigned int i=0;i<arr.getCount();++i)
        {
            out.push_back(arr[i]);
            out.right->set_true_size(out.rcount-1,arr.get_true_size(i));
        }
        out.true_size=front_f.true_size+back_f.true_size;
        return out;
    }
    else if (front_f.single())
    {
        out=back_f;
        for (int i=arr.getCount()-1;i>=0;--i)
        {
            out.push_front(arr[i]);
            out.left->set_true_size(out.lcount-1,arr.get_true_size(i));
        }
        if (front_f.lcount)
        {
            out.push_front((*front_f.left)[0]);
            out.left->set_true_size(out.lcount-1,front_f.left-
>get_true_size(0));
        }
        else
        {
            out.push_front((*front_f.right)[0]);
            out.left->set_true_size(out.lcount-1,front_f.right-
>get_true_size(0));
        }
        out.true_size=front_f.true_size+back_f.true_size;
        return out;
    }
    else if (back_f.single())
    {
        out=front_f;
        for (unsigned int i=0;i<arr.getCount();++i)
        {
            out.push_back(arr[i]);
            out.right->set_true_size(out.rcount-1,arr.get_true_size(i));
        }
        if (back_f.lcount)
        {
            out.push_back((*back_f.left)[0]);
            out.right->set_true_size(out.rcount-1,back_f.left-
>get_true_size(0));
        }
        else
        {
            out.push_back((*back_f.right)[0]);
            out.right->set_true_size(out.rcount-1,back_f.right-
>get_true_size(0));
        }
        out.true_size=front_f.true_size+back_f.true_size;
        return out;
    }
    else
    {
        out.left=front_f.left;
        out.lcount=front_f.lcount;
        out.right=back_f.right;
        out.rcount=back_f.rcount;

        Node<void_ref_ptr> tarr(4);
        unsigned int pos=0;
        unsigned int totalsz=
back_f.lcount+front_f.rcount+arr.getCount();//max 12
        unsigned int nodesz;
        do  //max 4 times
        {
            if (totalsz>4)
                nodesz=3;
            else if (totalsz<4)
                nodesz=totalsz;
            else //if (totalsz==4)
                nodesz=2;

            Node<T> tmp(nodesz);
            while (tmp.getCount()<nodesz)//loop called a maximum 12
times
            {
                if (pos<front_f.rcount)
                {
                    tmp.push_back((*front_f.right)[pos]);
                    ++pos;
                }
                else if (pos-front_f.rcount<arr.getCount())
                {
                    tmp.push_back(arr[pos-front_f.rcount]);
                    ++pos;
                }
                else //if (pos-front_f.rcount-
arr.getCount()<back_f.lcount)
                {
                    tmp.push_back((*back_f.left)[back_f.lcount-1-(pos-
front_f.rcount-arr.getCount())]);
                    ++pos;
                }
            }

            tarr.push_back(void_ref_ptr(tmp));
            totalsz-=nodesz;
        } while (totalsz>0);

        out.spine=null_ref_ptr<Finger<void_ref_ptr>
>(Finger<void_ref_ptr>());

        if (!front_f.spine.isNull()&&!back_f.spine.isNull())
            out.spine=(front_f.spine-
>concat(*(front_f.spine),*(back_f.spine),tarr));
        else if (!front_f.spine.isNull())
            out.spine=(front_f.spine-
>concat(*(front_f.spine),Finger<void_ref_ptr>(),tarr));
        else if (!back_f.spine.isNull())
            out.spine=(front_f.spine->concat(Finger<void_ref_ptr
>(),*(back_f.spine),tarr));
        else
            out.spine=(front_f.spine->concat(Finger<void_ref_ptr
>(),Finger<void_ref_ptr >(),tarr));

        out.true_size=front_f.true_size+back_f.true_size;
        return out;
    }
};

template <typename T>
ostream & Finger<T>::print_aux(ostream & out, const Printer<T> & pr)
const
{
    out << '<';
    for (int i = lcount-1; i >= 0; --i)
    {
        pr.print(out, (*left)[i]);
        if (i > 0)
            out << ", ";
    }

    if (!spine.isNull())
    {
        out << ' ';
        spine->print_aux(out, NodePrinter<T>(pr));
        out << ' ';
    }
    else if (lcount && rcount)
        out << " <> ";

    for (unsigned int i = 0; i < rcount; ++i)
    {
        if (i > 0)
            out << ", ";
        pr.print(out, (*right)[i]);
    }

    out << '>';
    return out;
};

#endif
```

Node.h

```
#ifndef _Node_h
#define _Node_h

template  <typename T>
class Node
{
    private:
    T *contents;
    unsigned int size;
    unsigned int count;
    unsigned int *total_elem;

    public:
    Node():size(1),count(0){contents=new T[1];total_elem=new unsigned
int[1];};//this is needed for ref_ptr
    Node(unsigned int sz):size(sz),count(0){contents=new
T[size];total_elem=new unsigned int[size];}
    Node(const Node &n):size(n.size),count(0){total_elem=new unsigned
int[size];contents=new T[size];for (unsigned int i=0;i<n.count;+
+i)push_back(n[i]);}
    Node(const Node &n,unsigned int sz):size(sz),count(0)
{total_elem=new unsigned int[size];contents=new T[size];push_back(n);}
    ~Node(){delete[] contents;delete[] total_elem;}

    unsigned int get_true_size(unsigned int index){return
total_elem[index];}
    void set_true_size(unsigned int index, unsigned int sz)
{total_elem[index]=sz;}

    Node & operator=(const Node &n)
    {
        delete[] contents;
        delete[] total_elem;
        size=n.size;
        contents=new T[size];
        total_elem=new unsigned int[size];
        push_back(n);
        return *this;
    }

    void push_back(const T &val)
    {
            contents[count]=val;
            total_elem[count]=1;
            ++count;
    }/
    void push_back(const Node &n)
    {
        if (n.count<size)
            for (unsigned int i=0;i<n.count;++i)
{push_back(n[i]);total_elem[i]=n.total_elem[i];}
        else
            for (unsigned int i=0;i<size;++i)
{push_back(n[i]);total_elem[i]=n.total_elem[i];}
```

```
    }
    T &operator[](unsigned int i) {if (i<count) return contents[i];}
    T operator[](unsigned int i)const {return contents[i];}
    unsigned int getCount()const{return count;}

};

#endif
```

refptr.h

```
#ifndef __REFPTR__
#define __REFPTR__


class _base_ref_rec {

    friend class void_ref_ptr;
      protected:
      int _ref_count;
      _base_ref_rec() : _ref_count(1) {}
      virtual ~_base_ref_rec(){}
};

template <typename T>
class ref_ptr;
template <typename T>
class null_ref_ptr;

template <typename T>
class _ref_rec : public _base_ref_rec {

      friend class ref_ptr<T>;
      friend class null_ref_ptr<T>;
    friend class void_ref_ptr;
      T _ref_value;
      _ref_rec():_base_ref_rec(){}
      _ref_rec(const T & arg) : _base_ref_rec(),_ref_value(arg) {}
};

class void_ref_ptr{

    _base_ref_rec *_ptr;
public:
      void_ref_ptr() : _ptr(new _ref_rec<int>()) {}
      template <typename T>
      void_ref_ptr(const T & arg) : _ptr(new _ref_rec<T>(arg)) {}
      void_ref_ptr(const void_ref_ptr & other) : _ptr(other._ptr) {
            _ptr->_ref_count++;
      }


      ~void_ref_ptr() {
            if (--_ptr->_ref_count == 0)
                  delete _ptr;
      }

      void_ref_ptr & operator=(const void_ref_ptr & other) {
            other._ptr->_ref_count++;     /* must be first! */
            if (--_ptr->_ref_count == 0)
                  delete _ptr;
            _ptr = other._ptr;
            return *this;
      }
    template <typename T>
    T & get() const { return ((_ref_rec<T>*)_ptr)->_ref_value; }
    template <typename T>
    T *get_ptr()const{return &(((_ref_rec<T>*)_ptr)->_ref_value);}
};


template <typename T>
class ref_ptr {

      _ref_rec<T> *_ptr;
public:
      ref_ptr() : _ptr(new _ref_rec<T>()) {}
      ref_ptr(const T & arg) : _ptr(new _ref_rec<T>(arg)) {}
      ref_ptr(const ref_ptr<T> & other) : _ptr(other._ptr) {
            _ptr->_ref_count++;
      }
      ~ref_ptr() {
            if (--_ptr->_ref_count == 0)
                  delete _ptr;
      }

      ref_ptr<T> & operator=(const ref_ptr<T> & other) {
            other._ptr->_ref_count++;     /* must be first! */
            if (--_ptr->_ref_count == 0)
                  delete _ptr;
            _ptr = other._ptr;
            return *this;
      }

      T & operator*() const { return _ptr->_ref_value; }
      T * operator->() const { return &(_ptr->_ref_value); }
};


template <typename T>
class null_ref_ptr {
      _ref_rec<T> *_ptr;
public:
      null_ref_ptr()  {_ptr=0;}
      null_ref_ptr(const T & arg) : _ptr(new _ref_rec<T>(arg)) {}
      null_ref_ptr(const null_ref_ptr<T> & other) : _ptr(other._ptr) {
            _ptr->_ref_count++;
      }

      ~null_ref_ptr() {
          if (_ptr)
          {
            if (--_ptr->_ref_count == 0)
                delete _ptr;
          }
      }

      null_ref_ptr<T> & operator=(const null_ref_ptr<T> & other) {
          if (other._ptr)
            other._ptr->_ref_count++;     /* must be first! */
        if (_ptr)
        {
            if (--_ptr->_ref_count == 0)
                delete _ptr;
        }
            _ptr = other._ptr;
            return *this;
      }
      bool isNull() const {if (_ptr)return false;else return true;}

      T & operator*() const { return _ptr->_ref_value; }
      T * operator->() const { return &(_ptr->_ref_value); }
};


#endif
```

Interactive environment, Main.cpp

```
#include "Node.h"
#include "Finger.h"
#include <iostream>
using namespace std;

const unsigned int max_size = 200;


int main()
{
    srand(1);
    int it=1, count=0;
    int random;
    Finger<int> f1,f2,f3;
    while (true)
    {
        cout <<"actions required, 0 to stop , -1 clears f1, -2 clears 
f2:";
        cin>>it;
        if (it==0)
            break;
        else if (it==-1)
        {
            while (f1.size()>0)
            {
                random=rand();
                if (random>(RAND_MAX/2))
                {
                    f1.pop_back();
                    cout<<"f1.pop_back()";
                }
                else
                {
                    f1.pop_front();
                    cout<<"f1.pop_front()";
                }

                cout <<" sucessful\n\nf1:";
                f1.print(cout);
                cout <<"\n\n";
            }
        }
        else if  (it==-2)
        {
            while (f2.size()>0)
            {
                random=rand();
                if (random>(RAND_MAX/2))
                {
                    f2.pop_back();
                    cout<<"f2.pop_back()";
                }
                else
                {
```

```
                f2.pop_front();
                cout<<"f2.pop_front()";
            }

            cout <<" sucessful\n\nf2:";
            f2.print(cout);
            cout <<"\n\n";
        }
    }
    else
    {
        for (int i=0;i<it;i++)
        {
            ++count;
            random=rand();
            cout <<"action number "<<count<<": ";


            switch (random%8)
            {
                case 0 :
                case 1 :
                case 2 :
                {
                    if (random>(RAND_MAX/2))
                    {
                        if (random % max_size >= f1.size())
                        {
                            f1.push_back(random);
                            cout<<"f1.push_back("<<random<<")";
                        }
                        else
                        {
                            f1.pop_back();
                            cout<<"f1.pop_back()";
                        }
                    }
                    else
                    {
                        if (random % max_size >= f2.size())
                        {
                            f2.push_back(random);
                            cout<<"f2.push_back("<<random<<")";
                        }
                        else
                        {
                            f2.pop_back();
                            cout<<"f2.pop_back()";
                        }
                    }
                    break;
                }

                case 3 :
                case 4 :
                case 5 :  {
                    if (random>(RAND_MAX/2))
                    {
                        if (random % max_size >= f1.size())
                        {
                            f1.push_front(random);
                            cout<<"f1.push_front("<<random<<")";
                        }
                        else
                        {
                            f1.pop_front();
                            cout<<"f1.pop_front()";
                        }
                    }
                    else
                    {
                        if (random % max_size >= f2.size())
                        {
                            f2.push_front(random);
                            cout<<"f2.push_front("<<random<<")";
                        }
                        else
                        {
                            f2.pop_front();
                            cout<<"f2.pop_front()";
                        }
                    }
                    break;
                }
                case 6 :
                {
                    if (random>(RAND_MAX/2))
                    {
                        f1=f1+f2;
                        cout<<"f1=f1+f2";
                    }
                    else
                    {
                      f2=f2+f1;
                      cout<<"f2=f2+f1";
                    }
                    break;
                }
                case 7 :    //immutability test
                {
                    if (random>(RAND_MAX/2))
                    {
                        f3=f1;
                        cout<<"f3=f1";
                    }
                    else
                    {
                      f3=f2;
                      cout<<"f3=f2";
                    }
                    break;
                }
            }
            cout <<" sucessful\n\nf1:";
```

```
                f1.print(cout);
                cout <<"\n\nf2:";
                f2.print(cout);
                cout <<"\n\nf3:";
                f3.print(cout);
                cout <<"\n\nf1.size="<<f1.size();
                cout <<"\n\nf2.size="<<f2.size();
                cout <<"\n\nf3.size="<<f3.size()<<"\n\n\n";
            }
        }
    }
    return 0;
}
```

Appendix B – Code Listings

Concatenation Test, Main.cpp

```
#include "Node.h"
#include "Finger.h"
#include <iostream>
#include <list>
using namespace std;

int main()
{
    srand(45);
    int random;
    Finger<int> f1,f2;
    list<int> l1,l2,l3;

    for (int i=0;i<6;i++)
    {
        random=rand();
        if (random>(RAND_MAX/2))
        {
            f1.push_front(random);
            l1.push_front(random);
        }
        else
        {
            f1.push_back(random);
            l1.push_back(random);
        }
    }
    l2=l1;
    f2=f1;
    for (int i=1;i<300000;i++)
    {
       l1.insert(l1.end(), l2.begin(),l2.end());
       f1=f1+f2;

       if (i%50000==0)
       {
            f2=f2+f2;
            l3=l2;
            l2.insert(l2.end(), l3.begin(),l3.end());
            f1=f2;
            l1=l2;
       }
    }


    return 0;
}
```

Queue Test, Main.cpp

```
#include "Node.h"
#include "Finger.h"
#include <list>

using namespace std;

int main()
{
    srand(45);
    int random;
    Finger<int> f1;
    list<int> l1;

    for (int i=0;i<1000;i++)
    {
        random=rand();
        if (random>(RAND_MAX/2))
        {
            f1.push_front(random);
            l1.push_front(random);
        }
        else
        {
            f1.push_back(random);
            l1.push_back(random);
        }
    }
    for (int i=0;i<10000000;i++)
    {
        random=rand();
        f1.push_front(random);
        l1.push_front(random);
        f1.pop_back();
        l1.pop_back();
    }
    for (int i=0;i<10000000;i++)
    {
        random=rand();
        f1.push_back(random);
        l1.push_back(random);
        f1.pop_front();
        l1.pop_front();
    }


    return 0;
}
```