Sorting and Searching Algorithms: A Cookbook

Thomas Niemann

Preface

This is a collection of algorithms for sorting and searching. Descriptions are brief and intuitive, with just enough theory thrown in to make you nervous. I assume you know C, and that you are familiar with concepts such as arrays and pointers.

The first section introduces basic data structures and notation. The next section presents several *sorting* algorithms. This is followed by techniques for implementing *dictionaries*, structures that allow efficient *search*, *insert*, and *delete* operations. The last section illustrates algorithms that sort data and implement dictionaries for very large files. Source code for each algorithm, in ANSI C, is available at the site listed below.

THOMAS NIEMANN
Portland, Oregon

email: thomasn@jps.net
home: http://members.xoom.com/thomasn/s_man.htm

By the same author:

A Guide to Lex and Yacc, at http://members.xoom.com/thomasn/y_man.htm.

CONTENTS

1. Introduction

Arrays and linked lists are two basic data structures used to store information. We may wish to *search*, *insert* or *delete* records in a database based on a key value. This section examines the performance of these operations on arrays and linked lists.

Arrays

Figure 1-1 shows an array, seven elements long, containing numeric values. To search the array *sequentially*, we may use the algorithm in Figure 1-2. The maximum number of comparisons is 7, and occurs when the key we are searching for is in *A*[6].

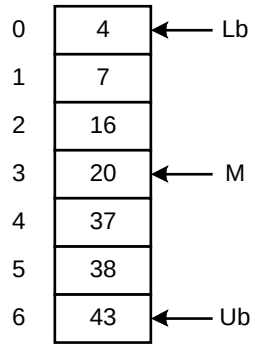

Figure 1-1: An Array

```
int function SequentialSearch (Array A , int Lb , int Ub , int Key );
    begin
    for i = Lb to Ub do
        if A [ i ] =  Key then
            return i ;
        return –1;
    end;
```

Figure 1-2: Sequential Search

```
int function BinarySearch (Array A , int Lb , int Ub , int Key );
    begin
    do forever
        M = ( Lb + Ub )/2;
        if ( Key < A[M]) then
            Ub = M – 1;
        else if (Key > A[M]) then
            Lb = M + 1;
        else
            return M ;
        if (Lb > Ub) then
            return –1;
    end;
```

Figure 1-3: Binary Search

If the data is sorted, a *binary* search may be done (Figure 1-3). Variables *Lb* and *Ub* keep track of the *lower bound* and *upper bound* of the array, respectively. We begin by examining the middle element of the array. If the key we are searching for is less than the middle element, then it must reside in the top half of the array. Thus, we set *Ub* to (*M* – 1). This restricts our next iteration through the loop to the top half of the array. In this way, each iteration halves the size of the array to be searched. For example, the first iteration will leave 3 items to test. After the second iteration, there will be one item left to test. Therefore it takes only three iterations to find any number.

This is a powerful method. Given an array of 1023 elements, we can narrow the search to 511 elements in one comparison. After another comparison, and we're looking at only 255 elements. In fact, we can search the entire array in only 10 comparisons.

In addition to searching, we may wish to insert or delete entries. Unfortunately, an array is not a good arrangement for these operations. For example, to insert the number 18 in Figure 1-1, we would need to shift *A*[3]…*A*[6] down by one slot. Then we could copy number 18 into *A*[3]. A similar problem arises when deleting numbers. To improve the efficiency of insert and delete operations, linked lists may be used.

Linked Lists

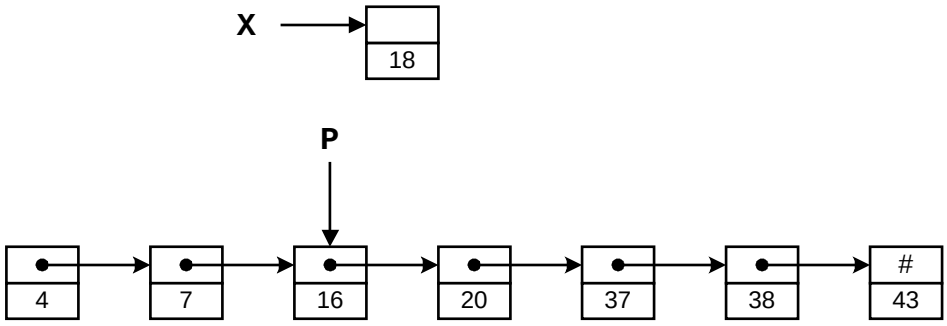

Figure 1-4: A Linked List

In Figure 1-4 we have the same values stored in a linked list. Assuming pointers **X** and **P**, as shown in the figure, value 18 may be inserted as follows:

```
X->Next = P->Next;
P->Next = X;
```

Insertion and deletion operations are very efficient using linked lists. You may be wondering how pointer **P** was set in the first place. Well, we had to do a sequential search to find the insertion point **X**. Although we improved our performance for insertion/deletion, it was done at the expense of search time.

Timing Estimates

Several methods may be used to compare the performance of algorithms. One way is simply to run several tests for each algorithm and compare the timings. Another way is to estimate the time required. For example, we may state that search time is $\boldsymbol{O}(n)$ (big-oh of n). This means that search time, for large n, is proportional to the number of items n in the list. Consequently, we would expect search time to triple if our list increased in size by a factor of three. The big-$\boldsymbol{O}$ notation does not describe the exact time that an algorithm takes, but only indicates an upper bound on execution time within a constant factor. If an algorithm takes $\boldsymbol{O}(n^2)$ time, then execution time grows no worse than the square of the size of the list.

n	lg n	n lg n	$n^{1.25}$	n^2
1	0	0	1	1
16	4	64	32	256
256	8	2,048	1,024	65,536
4,096	12	49,152	32,768	16,777,216
65,536	16	1,048,565	1,048,476	4,294,967,296
1,048,476	20	20,969,520	33,554,432	1,099,301,922,576
16,775,616	24	402,614,784	1,073,613,825	281,421,292,179,456

Table 1-1: Growth Rates

Table 1-1 illustrates growth rates for various functions. A growth rate of $\boldsymbol{O}(\lg n)$ occurs for algorithms similar to the binary search. The lg (logarithm, base 2) function increases by one when n is doubled. Recall that we can search twice as many items with one more comparison in the binary search. Thus the binary search is a $\boldsymbol{O}(\lg n)$ algorithm.

If the values in Table 1-1 represented microseconds, then a $\boldsymbol{O}(\lg n)$ algorithm may take 20 microseconds to process 1,048,476 items, a $\boldsymbol{O}(n^{1.25})$ algorithm might take 33 seconds, and a $\boldsymbol{O}(n^2)$ algorithm might take up to 12 days! In the following chapters a timing estimate for each algorithm, using big-$\boldsymbol{O}$ notation, will be included. For a more formal derivation of these formulas you may wish to consult the references.

Summary

As we have seen, sorted arrays may be searched efficiently using a binary search. However, we must have a sorted array to start with. In the next section various ways to sort arrays will be examined. It turns out that this is computationally expensive, and considerable research has been done to make sorting algorithms as efficient as possible.

Linked lists improved the efficiency of insert and delete operations, but searches were sequential and time-consuming. Algorithms exist that do all three operations efficiently, and they will be the discussed in the section on dictionaries.

2. Sorting

Several algorithms are presented, including *insertion sort*, *shell sort*, and *quicksort*. Sorting by insertion is the simplest method, and doesn't require any additional storage. Shell sort is a simple modification that improves performance significantly. Probably the most efficient and popular method is quicksort, and is the method of choice for large arrays.

2.1 Insertion Sort

One of the simplest methods to sort an array is an insertion sort. An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence. Both average and worst-case time is $\boldsymbol{O}(n^2)$. For further reading, consult Knuth [1998].

Theory

Starting near the top of the array in Figure 2-1(a), we extract the 3. Then the above elements are shifted down until we find the correct place to insert the 3. This process repeats in Figure 2-1(b) with the next number. Finally, in Figure 2-1(c), we complete the sort by inserting 2 in the correct place.

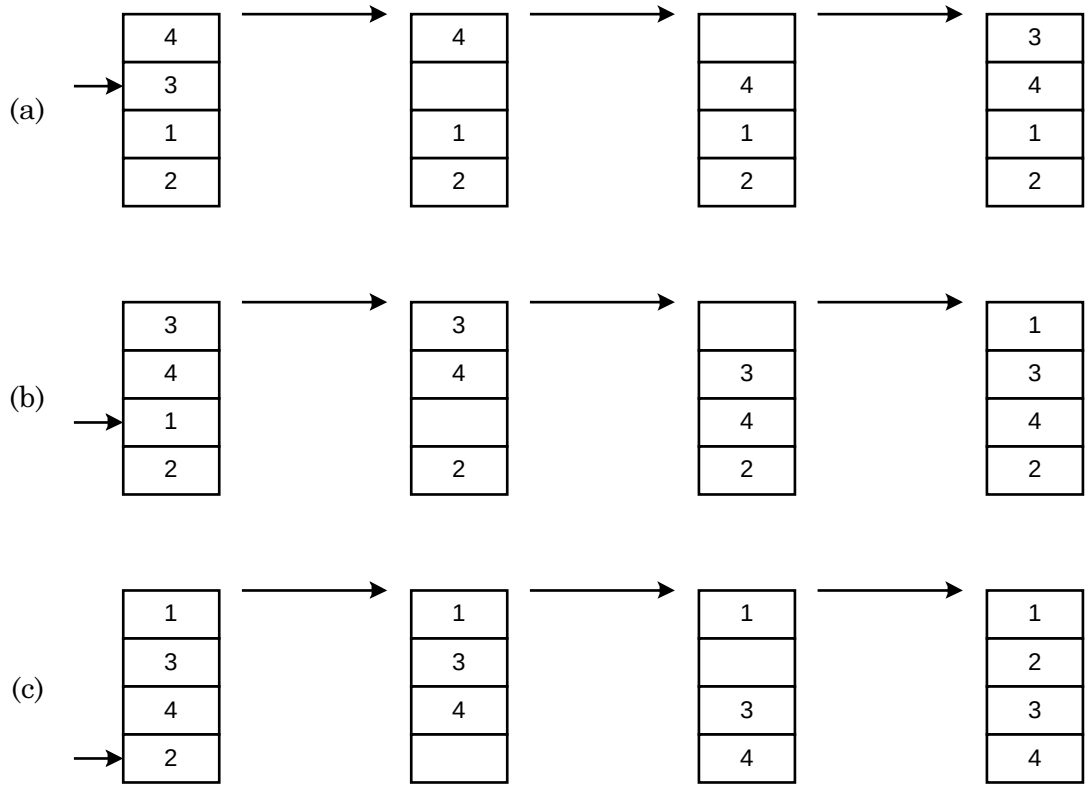

Figure 2-1: Insertion Sort

Assuming there are n elements in the array, we must index through $n - 1$ entries. For each entry, we may need to examine and shift up to $n - 1$ other entries, resulting in a $\boldsymbol{O}(n^2)$ algorithm. The insertion sort is an *in-place* sort. That is, we sort the array in-place. No extra memory is required. The insertion sort is also a *stable* sort. Stable sorts retain the original ordering of keys when identical keys are present in the input data.

Implementation

Source for the insertion sort algorithm may be found in file **`ins.c`**. Typedef **`T`** and comparison operator **`compGT`** should be altered to reflect the data stored in the table.

2.2 Shell Sort

Shell sort, developed by Donald L. Shell, is a non-stable in-place sort. Shell sort improves on the efficiency of insertion sort by *quickly* shifting values to their destination. Average sort time is $\boldsymbol{O}(n^{1.25})$, while worst-case time is $\boldsymbol{O}(n^{1.5})$. For further reading, consult Knuth [1998].

Theory

In Figure 2-2(a) we have an example of sorting by insertion. First we extract 1, shift 3 and 5 down one slot, and then insert the 1, for a count of 2 shifts. In the next frame, two shifts are required before we can insert the 2. The process continues until the last frame, where a total of 2 + 2 + 1 = 5 shifts have been made.

In Figure 2-2(b) an example of shell sort is illustrated. We begin by doing an insertion sort using a *spacing* of two. In the first frame we examine numbers 3-1. Extracting 1, we shift 3 down one slot for a shift count of 1. Next we examine numbers 5-2. We extract 2, shift 5 down, and then insert 2. After sorting with a spacing of two, a final pass is made with a spacing of one. This is simply the traditional insertion sort. The total shift count using shell sort is 1+1+1 = 3. By using an initial spacing larger than one, we were able to quickly shift values to their proper destination.

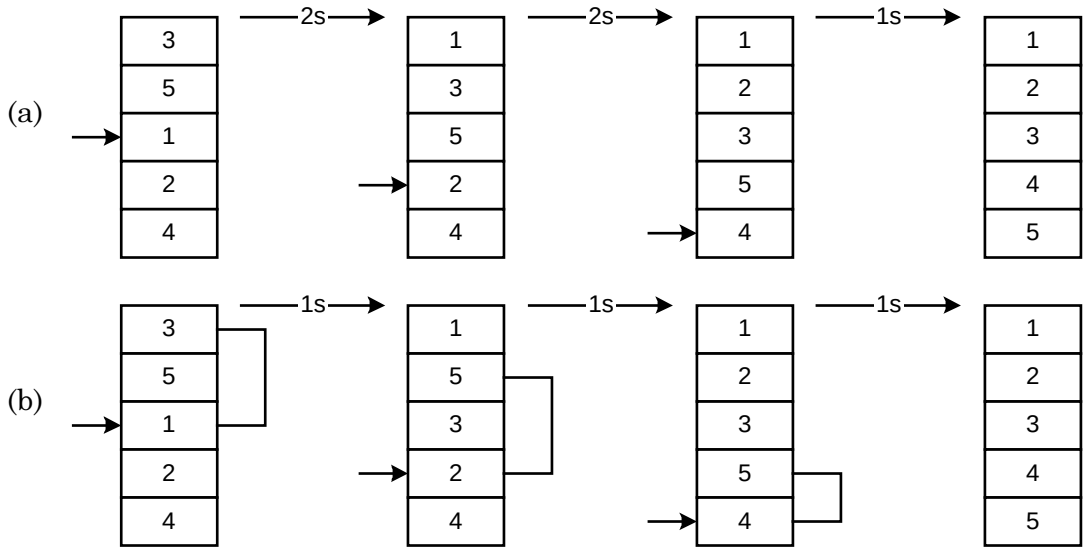

Figure 2-2: Shell Sort

Various spacings may be used to implement shell sort. Typically the array is sorted with a large spacing, the spacing reduced, and the array sorted again. On the final sort, spacing is one. Although the shell sort is easy to comprehend, formal analysis is difficult. In particular, optimal spacing values elude theoreticians. Knuth has experimented with several values and recommends that spacing *h* for an array of size *N* be based on the following formula:

$$\text{Let } h_1 = 1,\ h_{s+1} = 3h_s + 1,\ \text{and stop with } h_t \text{ when } h_{t+2} \geq N$$

Thus, values of h are computed as follows:

$$h_1 = 1$$
$$h_2 = (3 \times 1) + 1 = 4$$
$$h_3 = (3 \times 4) + 1 = 13$$
$$h_4 = (3 \times 13) + 1 = 40$$
$$h_5 = (3 \times 40) + 1 = 121$$

To sort 100 items we first find h_s such that $h_s \geq 100$. For 100 items, h_5 is selected. Our final value (h_t) is two steps lower, or h_3. Therefore our sequence of h values will be 13-4-1. Once the initial h value has been determined, subsequent values may be calculated using the formula

$$h_{s-1} = h_s / 3$$

Implementation

Source for the shell sort algorithm may be found in file **`shl.c`**. Typedef **T** and comparison operator **`compGT`** should be altered to reflect the data stored in the array. The central portion of the algorithm is an insertion sort with a spacing of **`h`**.

2.3 Quicksort

Although the shell sort algorithm is significantly better than insertion sort, there is still room for improvement. One of the most popular sorting algorithms is quicksort. Quicksort executes in $O(n \lg n)$ on average, and $O(n^2)$ in the worst-case. However, with proper precautions, worst-case behavior is very unlikely. Quicksort is a non-stable sort. It is not an in-place sort as stack space is required. For further reading, consult Cormen [1990].

Theory

The quicksort algorithm works by partitioning the array to be sorted, then recursively sorting each partition. In *Partition* (Figure 2-3), one of the array elements is selected as a pivot value. Values smaller than the pivot value are placed to the left of the pivot, while larger values are placed to the right.

```
int function Partition (Array A, int Lb, int Ub);
    begin
    select a pivot from A[Lb]…A[Ub];
    reorder A[Lb]…A[Ub] such that:
        all values to the left of the pivot are ≤ pivot
        all values to the right of the pivot are ≥ pivot
    return pivot position;
    end;

procedure QuickSort (Array A, int Lb, int Ub);
    begin
    if Lb < Ub then
        M = Partition (A, Lb, Ub);
        QuickSort (A, Lb, M – 1);
        QuickSort (A, M + 1, Ub);
    end;
```

Figure 2-3: Quicksort Algorithm

In Figure 2-4(a), the pivot selected is 3. Indices are run starting at both ends of the array. One index starts on the left and selects an element that is larger than the pivot, while another index starts on the right and selects an element that is smaller than the pivot. In this case, numbers 4 and 1 are selected. These elements are then exchanged, as is shown in Figure 2-4(b). This process repeats until all elements to the left of the pivot are ≤ the pivot, and all items to the right of the pivot are ≥ the pivot. *QuickSort* recursively sorts the two sub-arrays, resulting in the array shown in Figure 2-4(c).

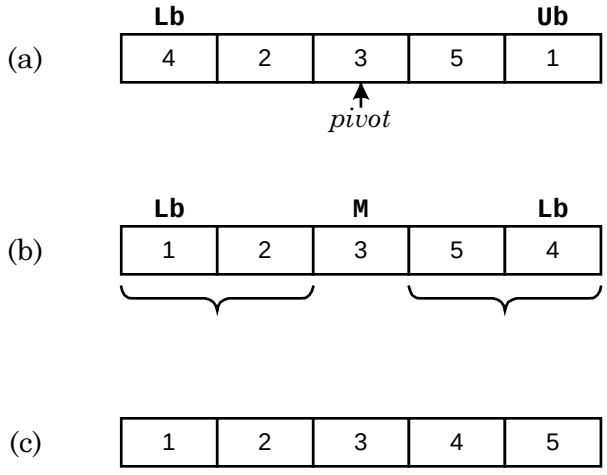

Figure 2-4: Quicksort Example

As the process proceeds, it may be necessary to move the pivot so that correct ordering is maintained. In this manner, *QuickSort* succeeds in sorting the array. If we're lucky the pivot selected will be the median of all values, equally dividing the array. For a moment, let's assume

that this is the case. Since the array is split in half at each step, and *Partition* must eventually examine all *n* elements, the run time is $O(n \lg n)$.

To find a pivot value, *Partition* could simply select the first element (*A[Lb]*). All other values would be compared to the pivot value, and placed either to the left or right of the pivot as appropriate. However, there is one case that fails miserably. Suppose the array was originally in order. *Partition* would always select the lowest value as a pivot and split the array with one element in the left partition, and *Ub – Lb* elements in the other. Each recursive call to quicksort would only diminish the size of the array to be sorted by one. Therefore *n* recursive calls would be required to do the sort, resulting in a $O(n^2)$ run time. One solution to this problem is to randomly select an item as a pivot. This would make it *extremely* unlikely that worst-case behavior would occur.

Implementation

The source for the quicksort algorithm may be found in file **`qui.c`**. Typedef **`T`** and comparison operator **`compGT`** should be altered to reflect the data stored in the array. Several enhancements have been made to the basic quicksort algorithm:

- The center element is selected as a pivot in **`partition`**. If the list is partially ordered, this will be a good choice. Worst-case behavior occurs when the center element happens to be the largest or smallest element each time **`partition`** is invoked.

- For short arrays, **`insertSort`** is called. Due to recursion and other overhead, quicksort is not an efficient algorithm to use on small arrays. Consequently, any array with fewer than 12 elements is sorted using an insertion sort. The optimal cutoff value is not critical and varies based on the quality of generated code.

- Tail recursion occurs when the last statement in a function is a call to the function itself. Tail recursion may be replaced by iteration, resulting in a better utilization of stack space. This has been done with the second call to *QuickSort* in Figure 2-3.

- After an array is partitioned, the smallest partition is sorted first. This results in a better utilization of stack space, as short partitions are quickly sorted and dispensed with.

Included in file **`qsort.c`** is the source for **`qsort`**, an ANSI-C standard library function usually implemented with quicksort. Recursive calls were replaced by explicit stack operations. Table 2-1 shows timing statistics and stack utilization before and after the enhancements were applied.

	time (μs)		*stacksize*	
count	*before*	*after*	*before*	*after*
16	103	51	540	28
256	1,630	911	912	112
4,096	34,183	20,016	1,908	168
65,536	658,003	470,737	2,436	252

Table 2-1: Effect of Enhancements on Speed and Stack Utilization

2.4 Comparison

In this section we will compare the sorting algorithms covered: insertion sort, shell sort, and quicksort. There are several factors that influence the choice of a sorting algorithm:

- *Stable sort*. Recall that a stable sort will leave identical keys in the same relative position in the sorted output. Insertion sort is the only algorithm covered that is stable.

- *Space*. An in-place sort does not require any extra space to accomplish its task. Both insertion sort and shell sort are in-place sorts. Quicksort requires stack space for recursion, and therefore is not an in-place sort. Tinkering with the algorithm considerably reduced the amount of time required.

- *Time*. The time required to sort a dataset can easily become astronomical (Table 1-1). Table 2-2 shows the relative timings for each method. The time required to sort a randomly ordered dataset is shown in Table 2-3.

- *Simplicity*. The number of statements required for each algorithm may be found in Table 2-2. Simpler algorithms result in fewer programming errors.

method	*statements*	*average time*	*worst-case time*
insertion sort	9	$\boldsymbol{O}(n^2)$	$\boldsymbol{O}(n^2)$
shell sort	17	$\boldsymbol{O}(n^{1.25})$	$\boldsymbol{O}(n^{1.5})$
quicksort	21	$\boldsymbol{O}(n \lg n)$	$\boldsymbol{O}(n^2)$

Table 2-2: Comparison of Methods

count	*insertion*	*shell*	*quicksort*
16	39 µs	45 µs	51 µs
256	4,969 µs	1,230 µs	911 µs
4,096	1.315 sec	.033 sec	.020 sec
65,536	416.437 sec	1.254 sec	.461 sec

Table 2-3: Sort Timings

3. Dictionaries

Dictionaries are data structures that support *search*, *insert*, and *delete* operations. One of the most effective representations is a *hash table*. Typically, a simple function is applied to the key to determine its place in the dictionary. Also included are *binary trees* and *red-black trees*. Both *tree* methods use a technique similar to the binary search algorithm to minimize the number of comparisons during search and update operations on the dictionary. Finally, *skip lists* illustrate a simple approach that utilizes random numbers to construct a dictionary.

3.1 Hash Tables

Hash tables are a simple and effective method to implement dictionaries. Average time to search for an element is $O(1)$, while worst-case time is $O(n)$. Cormen [1990] and Knuth [1998] both contain excellent discussions on hashing.

Theory

A hash table is simply an array that is addressed via a hash function. For example, in Figure 3-1, **`HashTable`** is an array with 8 elements. Each element is a pointer to a linked list of numeric data. The hash function for this example simply divides the data key by 8, and uses the remainder as an index into the table. This yields a number from 0 to 7. Since the range of indices for **`HashTable`** is 0 to 7, we are guaranteed that the index is valid.

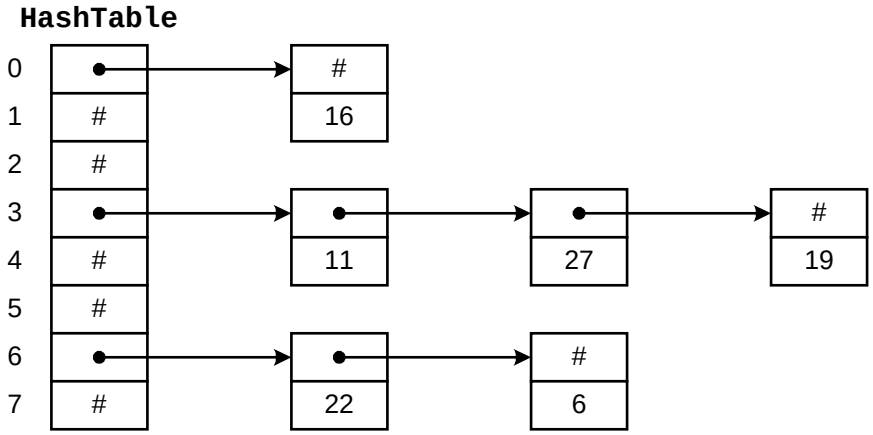

Figure 3-1: A Hash Table

To insert a new item in the table, we hash the key to determine which list the item goes on, and then insert the item at the beginning of the list. For example, to insert 11, we divide 11 by 8 giving a remainder of 3. Thus, 11 goes on the list starting at **`HashTable[3].`** To find a

number, we hash the number and chain down the correct list to see if it is in the table. To delete a number, we find the number and remove the node from the linked list.

Entries in the hash table are dynamically allocated and entered on a linked list associated with each hash table entry. This technique is known as *chaining*. An alternative method, where all entries are stored in the hash table itself, is known as direct or open addressing and may be found in the references.

If the hash function is uniform, or equally distributes the data keys among the hash table indices, then hashing effectively subdivides the list to be searched. Worst-case behavior occurs when all keys hash to the same index. Then we simply have a single linked list that must be sequentially searched. Consequently, it is important to choose a good hash function. Several methods may be used to hash key values. To illustrate the techniques, I will assume *unsigned char* is 8-bits, *unsigned short int* is 16-bits, and *unsigned long int* is 32-bits.

- *Division method (tablesize = prime).* This technique was used in the preceding example. A **`HashValue`**, from 0 to (**`HashTableSize - 1`**), is computed by dividing the key value by the size of the hash table and taking the remainder. For example:

```
typedef int HashIndexType;

HashIndexType Hash(int Key) {
    return Key % HashTableSize;
}
```

  Selecting an appropriate **`HashTableSize`** is important to the success of this method. For example, a **`HashTableSize`** of two would yield even hash values for even **`Keys`**, and odd hash values for odd **`Keys`**. This is an undesirable property, as all keys would hash to the same value if they happened to be even. If **`HashTableSize`** is a power of two, then the hash function simply selects a subset of the **`Key`** bits as the table index. To obtain a more random scattering, **`HashTableSize`** should be a prime number not too close to a power of two.

- *Multiplication method (tablesize = 2^n).* The multiplication method may be used for a **`HashTableSize`** that is a power of 2. The **`Key`** is multiplied by a constant, and then the necessary bits are extracted to index into the table. Knuth recommends using the fractional part of the product of the **`key`** and the golden ratio, or $\left(\sqrt{5}-1\right)/2$. For example, assuming a word size of 8 bits, the golden ratio is multiplied by 2^8 to obtain 158. The product of the 8-bit **`key`** and 158 results in a 16-bit integer. For a table size of 2^5 the 5 most significant bits of the least significant word are extracted for the hash value. The following definitions may be used for the multiplication method:

```
/* 8-bit index */
typedef unsigned char HashIndexType;
static const HashIndexType K = 158;

/* 16-bit index */
typedef unsigned short int HashIndexType;
static const HashIndexType K = 40503;

/* 32-bit index */
typedef unsigned long int HashIndexType;
static const HashIndexType K = 2654435769;

/* w=bitwidth(HashIndexType), size of table=2**m */
static const int S = w - m;
HashIndexType HashValue = (HashIndexType)(K * Key) >> S;
```

For example, if **HashTableSize** is 1024 (2^{10}), then a 16-bit index is sufficient and **S** would be assigned a value of 16 – 10 = 6. Thus, we have:

```
typedef unsigned short int HashIndexType;

HashIndexType Hash(int Key) {
    static const HashIndexType K = 40503;
    static const int S = 6;
    return (HashIndexType)(K * Key) >> S;
}
```

- *Variable string addition method (tablesize = 256).* To hash a variable-length string, each character is added, modulo 256, to a total. A **HashValue**, range 0-255, is computed.

```
typedef unsigned char HashIndexType;

HashIndexType Hash(char *str) {
    HashIndexType h = 0;
    while (*str) h += *str++;
    return h;
}
```

- *Variable string exclusive-or method (tablesize = 256).* This method is similar to the addition method, but successfully distinguishes similar words and anagrams. To obtain a hash value in the range 0-255, all bytes in the string are exclusive-or'd together. However, in the process of doing each exclusive-or, a random component is introduced.

```
typedef unsigned char HashIndexType;
unsigned char Rand8[256];

HashIndexType Hash(char *str) {
    unsigned char h = 0;
    while (*str) h = Rand8[h ^ *str++];
    return h;
}
```

Rand8 is a table of 256 8-bit unique random numbers. The exact ordering is not critical. The exclusive-or method has its basis in cryptography, and is quite effective Pearson [1990].

- *Variable string exclusive-or method (tablesize ≤ 65536).* If we hash the string twice, we may derive a hash value for an arbitrary table size up to 65536. The second time the string is hashed, one is added to the first character. Then the two 8-bit hash values are concatenated together to form a 16-bit hash value.

```
typedef unsigned short int HashIndexType;
unsigned char Rand8[256];

HashIndexType Hash(char *str) {
    HashIndexType h;
    unsigned char h1, h2;

    if (*str == 0) return 0;
    h1 = *str; h2 = *str + 1;
    str++;
    while (*str) {
        h1 = Rand8[h1 ^ *str];
        h2 = Rand8[h2 ^ *str];
        str++;
    }

    /* h is in range 0..65535 */
    h = ((HashIndexType)h1 << 8)|(HashIndexType)h2;
    /* use division method to scale */
    return h % HashTableSize
}
```

Assuming *n* data items, the hash table size should be large enough to accommodate a reasonable number of entries. As seen in Table 3-1, a small table size substantially increases the average time to find a key. A hash table may be viewed as a collection of linked lists. As the table becomes larger, the number of lists increases, and the average number of nodes on each list decreases. If the table size is 1, then the table is really a single linked list of length *n*. Assuming a perfect hash function, a table size of 2 has two lists of length *n*/2. If the table size is 100, then we have 100 lists of length *n*/100. This considerably reduces the length of the list to be searched. There is considerable leeway in the choice of table size.

size	*time*	*size*	*time*
1	869	128	9
2	432	256	6
4	214	512	4
8	106	1024	4
16	54	2048	3
32	28	4096	3
64	15	8192	3

Table 3-1: `HashTableSize` vs. Average Search Time (μs), 4096 entries

Implementation

Source for the hash table algorithm may be found in file **`has.c`**. Typedef **`T`** and comparison operator **`compEQ`** should be altered to reflect the data stored in the table. The **`hashTableSize`** must be determined and the **`hashTable`** allocated. The division method was used in the **`hash`** function. Function **`insertNode`** allocates a new node and inserts it in the table. Function **`deleteNode`** deletes and frees a node from the table. Function **`findNode`** searches the table for a particular value.

3.2 Binary Search Trees

In the Introduction, we used the binary search algorithm to find data stored in an array. This method is very effective, as each iteration reduced the number of items to search by one-half. However, since data was stored in an array, insertions and deletions were not efficient. Binary search trees store data in nodes that are linked in a tree-like fashion. For randomly inserted data, search time is $O(\lg n)$. Worst-case behavior occurs when ordered data is inserted. In this case the search time is $O(n)$. See Cormen [1990] for a more detailed description.

Theory

A binary search tree is a tree where each node has a left and right child. Either child, or both children, may be missing. Figure 3-2 illustrates a binary search tree. Assuming *k* represents the value of a given node, then a binary search tree also has the following property: all children to the left of the node have values smaller than *k*, and all children to the right of the node have values larger than *k*. The top of a tree is known as the *root*, and the exposed nodes at the bottom are known as *leaves*. In Figure 3-2, the root is node 20 and the leaves are nodes 4, 16, 37, and 43. The *height* of a tree is the length of the longest path from root to leaf. For this example the tree height is 2.

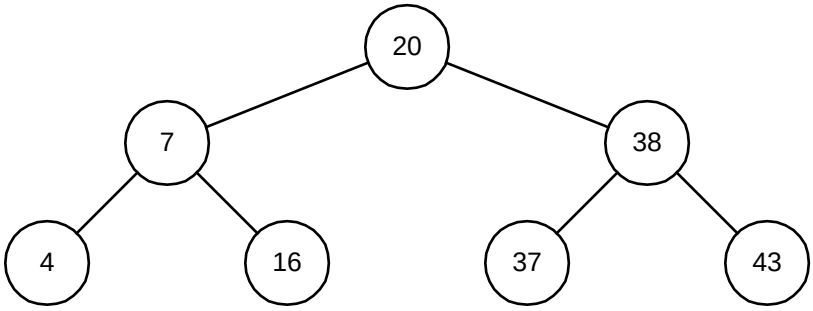

Figure 3-2: A Binary Search Tree

To search a tree for a given value, we start at the root and work down. For example, to search for 16, we first note that 16 < 20 and we traverse to the left child. The second comparison finds that 16 > 7, so we traverse to the right child. On the third comparison, we succeed.

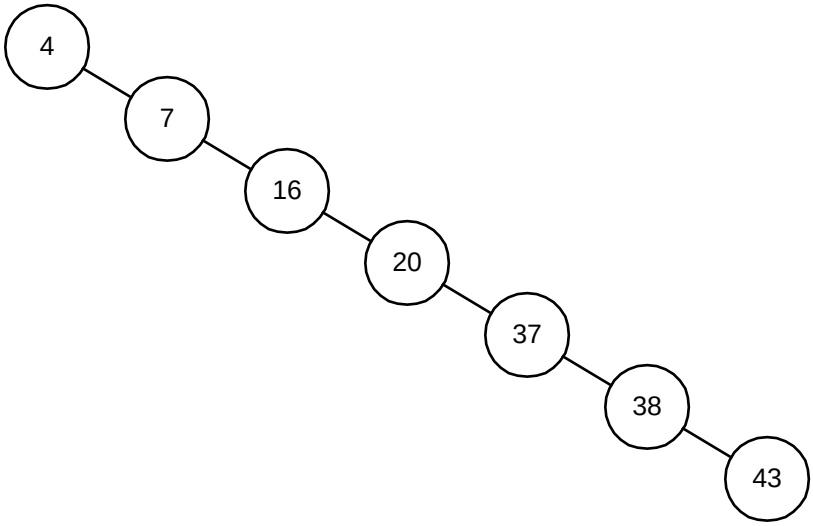

Figure 3-3: An Unbalanced Binary Search Tree

Each comparison results in reducing the number of items to inspect by one-half. In this respect, the algorithm is similar to a binary search on an array. However, this is true only if the tree is balanced. Figure 3-3 shows another tree containing the same values. While it is a binary search tree, its behavior is more like that of a linked list, with search time increasing proportional to the number of elements stored.

Insertion and Deletion

Let us examine insertions in a binary search tree to determine the conditions that can cause an unbalanced tree. To insert an 18 in the tree in Figure 3-2, we first search for that number. This causes us to arrive at node 16 with nowhere to go. Since 18 > 16, we simply add node 18 to the right child of node 16 (Figure 3-4).

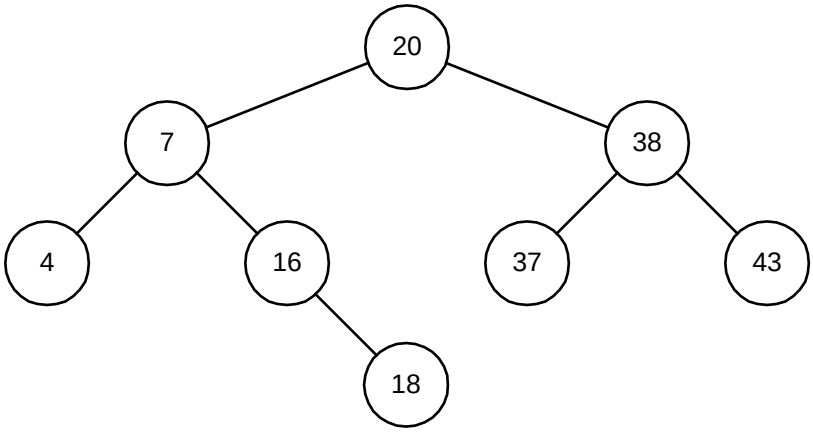

Figure 3-4: Binary Tree After Adding Node 18

Now we can see how an unbalanced tree can occur. If the data is presented in an ascending sequence, each node will be added to the right of the previous node. This will create one long chain, or linked list. However, if data is presented for insertion in a random order, then a more balanced tree is possible.

Deletions are similar, but require that the binary search tree property be maintained. For example, if node 20 in Figure 3-4 is removed, it must be replaced by node 37. This results in the tree shown in Figure 3-5. The rationale for this choice is as follows. The successor for node 20 must be chosen such that all nodes to the right are larger. Therefore we need to select the smallest valued node to the right of node 20. To make the selection, chain once to the right (node 38), and then chain to the left until the last node is found (node 37). This is the successor for node 20.

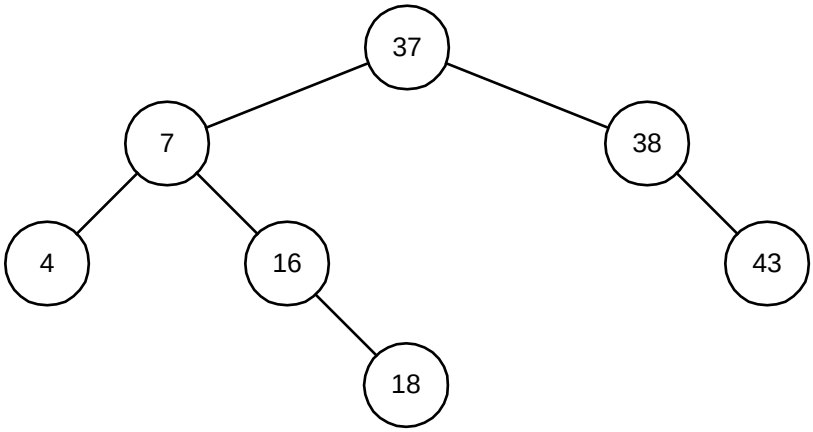

Figure 3-5: Binary Tree After Deleting Node 20

Implementation

Source for the binary search tree algorithm may be found in file **`bin.c`**. Typedef **`T`** and comparison operators **`compLT`** and **`compEQ`** should be altered to reflect the data stored in the tree. Each **`Node`** consists of **`left`**, **`right`**, and **`parent`** pointers designating each child and the parent. Data is stored in the **`data`** field. The tree is based at **`root`**, and is initially **`NULL`**. Function **`insertNode`** allocates a new node and inserts it in the tree. Function **`deleteNode`** deletes and frees a node from the tree. Function **`findNode`** searches the tree for a particular value.

3.3 Red-Black Trees

Binary search trees work best when they are balanced or the path length from root to any leaf is within some bounds. The red-black tree algorithm is a method for balancing trees. The name derives from the fact that each node is colored *red* or *black*, and the color of the node is instrumental in determining the balance of the tree. During insert and delete operations, nodes may be rotated to maintain tree balance. Both average and worst-case search time is $O(\lg n)$. See Cormen [1990] for details.

Theory

A red-black tree is a balanced binary search tree with the following properties:

1. Every node is colored red or black.
2. Every leaf is a **`NIL`** node, and is colored black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

The number of black nodes on a path from root to leaf is known as the *black height* of a tree. These properties guarantee that any path from the root to a leaf is no more than twice as long as any other path. To see why this is true, consider a tree with a black height of two. The shortest distance from root to leaf is two, where both nodes are black. The longest distance from root to leaf is four, where the nodes are colored (root to leaf): red, black, red, black. It is not possible to insert more black nodes as this would violate property 4, the black-height requirement. Since red nodes must have black children (property 3), having two red nodes in a row is not allowed. The largest path we can construct consists of an alternation of red-black nodes, or twice the length of a path containing only black nodes. All operations on the tree must maintain the properties listed above. In particular, operations that insert or delete items from the tree must abide by these rules.

Insertion

To insert a node, we search the tree for an insertion point, and add the node to the tree. The new node replaces an existing **`NIL`** node at the bottom of the tree, and has two **`NIL`** nodes as children. In the implementation, a **`NIL`** node is simply a pointer to a common *sentinel* node that is colored black. After insertion, the new node is colored red. Then the parent of the node is examined to determine if the red-black tree properties have been violated. If necessary, we recolor the node and do rotations to balance the tree.

By inserting a red node with two **`NIL`** children, we have preserved black-height property (property 4). However, property 3 may be violated. This property states that both children of a red node must be black. Although both children of the new node are black (they're **`NIL`**), consider the case where the parent of the new node is red. Inserting a red node under a red parent would violate this property. There are two cases to consider:

- *Red parent, red uncle*: Figure 3-6 illustrates a red-red violation. Node X is the newly inserted node, with both parent and uncle colored red. A simple recoloring removes the red-red violation. After recoloring, the grandparent (node B) must be checked for validity, as its parent may be red. Note that this has the effect of propagating a red node up the tree. On completion, the root of the tree is marked black. If it was originally red, then this has the effect of increasing the black-height of the tree.

- *Red parent, black uncle*: Figure 3-7 illustrates a red-red violation, where the uncle is colored black. Here the nodes may be rotated, with the subtrees adjusted as shown. At this point the algorithm may terminate as there are no red-red conflicts and the top of the subtree (node A) is colored black. Note that if node X was originally a right child, a left rotation would be done first, making the node a left child.

Each adjustment made while inserting a node causes us to travel up the tree one step. At most one rotation (2 if the node is a right child) will be done, as the algorithm terminates in this case. The technique for deletion is similar.

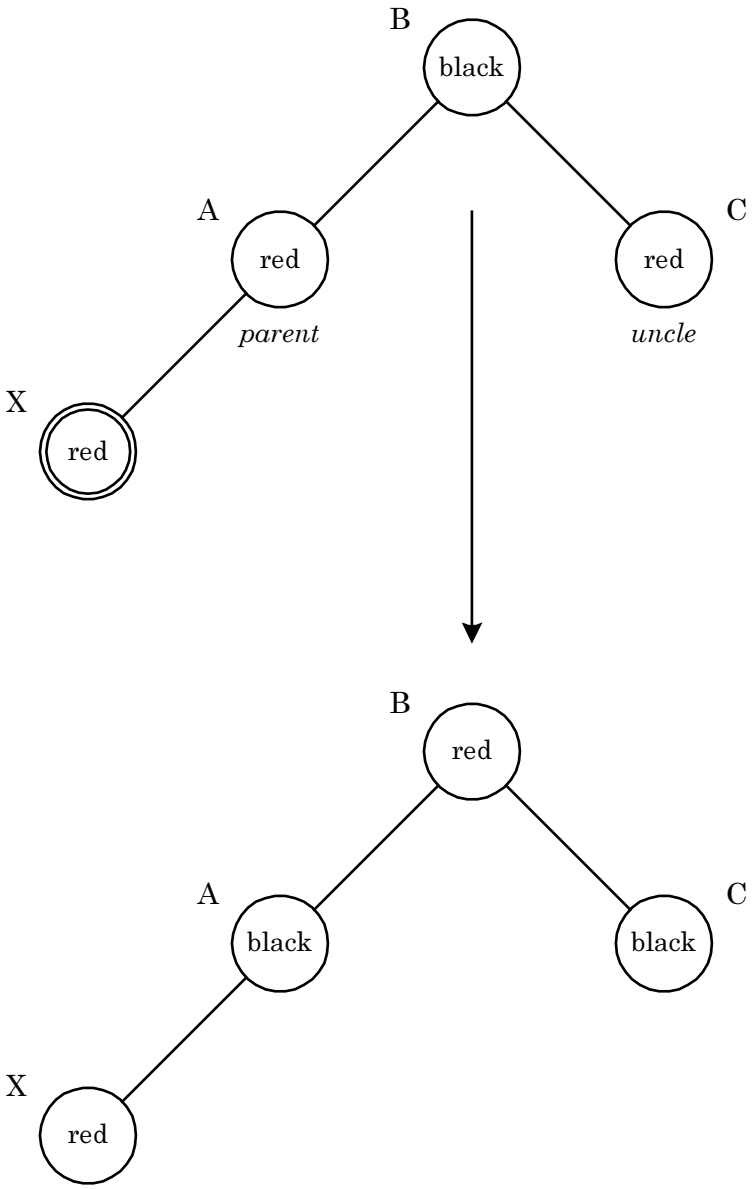

Figure 3-6: Insertion – Red Parent, Red Uncle

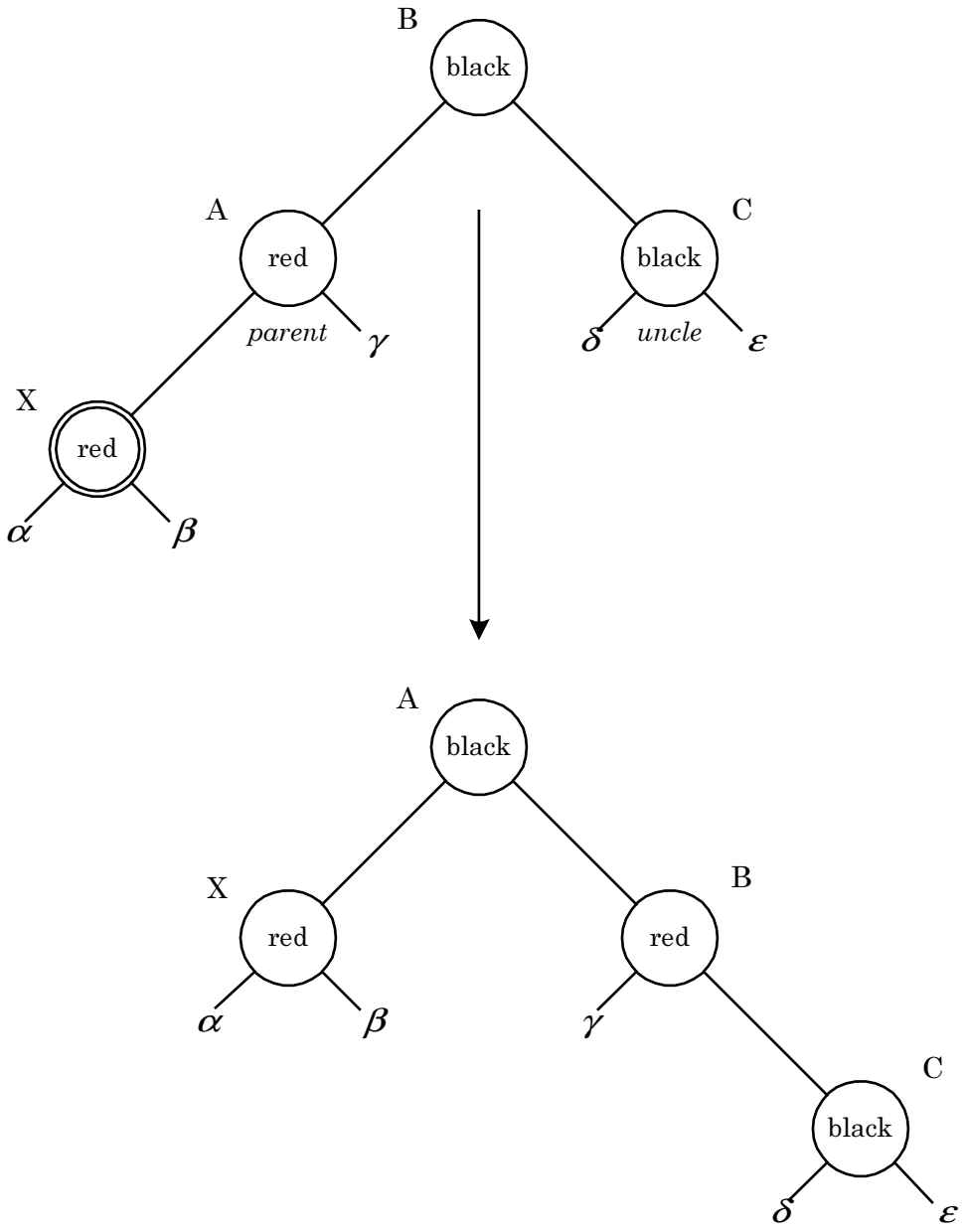

Figure 3-7: Insertion – Red Parent, Black Uncle

Implementation

Source for the red-black tree algorithm may be found in file **`rbt.c`**. Typedef **`T`** and comparison operators **`compLT`** and **`compEQ`** should be altered to reflect the data stored in the tree. Each **`Node`** consists of **`left`**, **`right`**, and **`parent`** pointers designating each child and the parent. The node color is stored in **`color`**, and is either **`RED`** or **`BLACK`**. The data is stored in the **`data`** field. All leaf nodes of the tree are **`sentinel`** nodes, to simplify coding. The tree is based at **`root`**, and initially is a **`sentinel`** node.

Function **`insertNode`** allocates a new node and inserts it in the tree. Subsequently, it calls **`insertFixup`** to ensure that the red-black tree properties are maintained. Function **`deleteNode`** deletes a node from the tree. To maintain red-black tree properties, **`deleteFixup`** is called. Function **`findNode`** searches the tree for a particular value.

3.4 Skip Lists

Skip lists are linked lists that allow you to *skip* to the correct node. The performance bottleneck inherent in a sequential scan is avoided, while insertion and deletion remain relatively efficient. Average search time is $O(\lg n)$. Worst-case search time is $O(n)$, but is extremely unlikely. An excellent reference for skip lists is Pugh [1990].

Theory

The indexing scheme employed in skip lists is similar in nature to the method used to lookup names in an address book. To lookup a name, you index to the tab representing the first character of the desired entry. In Figure 3-8, for example, the top-most list represents a simple linked list with no tabs. Adding tabs (middle figure) facilitates the search. In this case, level-1 pointers are traversed. Once the correct segment of the list is found, level-0 pointers are traversed to find the specific entry.

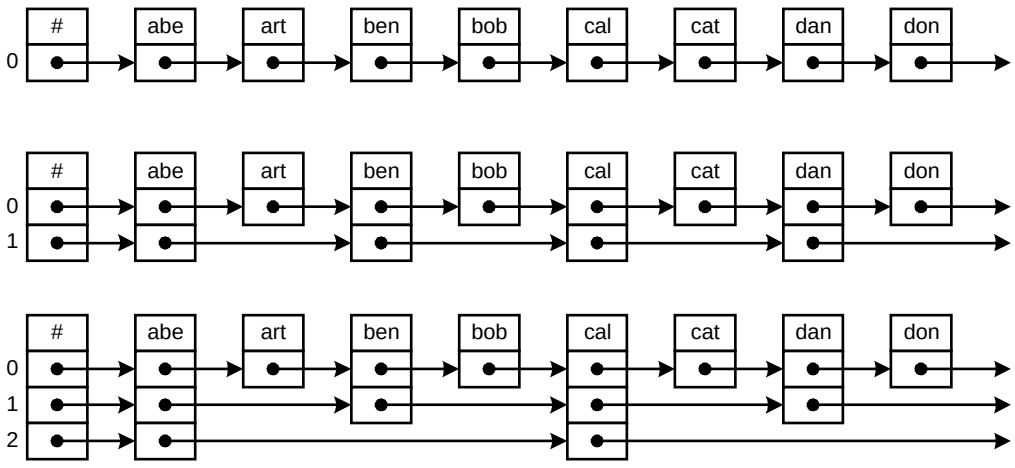

Figure 3-8: Skip List Construction

The indexing scheme may be extended as shown in the bottom figure, where we now have an index to the index. To locate an item, level-2 pointers are traversed until the correct segment of the list is identified. Subsequently, level-1 and level-0 pointers are traversed.

During insertion the number of pointers required for a new node must be determined. This is easily resolved using a probabilistic technique. A random number generator is used to toss a computer *coin*. When inserting a new node, the coin is tossed to determine if it should be level-1. If you win, the coin is tossed again to determine if the node should be level-2. Another win, and the coin is tossed to determine if the node should be level-3. This process repeats until you lose. If only one level (level-0) is implemented, the data structure is a simple linked-list with $\boldsymbol{O}(n)$ search time. However, if sufficient levels are implemented, the skip list may be viewed as a tree with the root at the highest level, and search time is $\boldsymbol{O}(\lg n)$.

The skip list algorithm has a probabilistic component, and thus a probabilistic bounds on the time required to execute. However, these bounds are quite tight in normal circumstances. For example, to search a list containing 1000 items, the probability that search time will be 5 times the average is about 1 in 1,000,000,000,000,000,000.

Implementation

Source for the skip list algorithm may be found in file **`skl.c`**. Typedef **`T`** and comparison operators **`compLT`** and **`compEQ`** should be altered to reflect the data stored in the list. In addition, **`MAXLEVEL`** should be set based on the maximum size of the dataset.

To initialize, **`initList`** is called. The list header is allocated and initialized. To indicate an empty list, all levels are set to point to the header. Function **`insertNode`** allocates a new node, searches for the correct insertion point, and inserts it in the list. While searching, the **`update`** array maintains pointers to the upper-level nodes encountered. This information is subsequently used to establish correct links for the newly inserted node. The **`newLevel`** is determined using a random number generator, and the node allocated. The forward links are then established using information from the **`update`** array. Function **`deleteNode`** deletes and frees a node, and is implemented in a similar manner. Function **`findNode`** searches the list for a particular value.

3.5 Comparison

We have seen several ways to construct dictionaries: hash tables, unbalanced binary search trees, red-black trees, and skip lists. There are several factors that influence the choice of an algorithm:

- *Sorted output*. If sorted output is required, then hash tables are not a viable alternative. Entries are stored in the table based on their hashed value, with no other ordering. For binary trees, the story is different. An in-order tree walk will produce a sorted list. For example:

```
void WalkTree(Node *P) {
    if (P == NIL) return;
    WalkTree(P->Left);

    /* examine P->Data here */

    WalkTree(P->Right);
}
WalkTree(Root);
```

To examine skip list nodes in order, simply chain through the level-0 pointers. For example:

```
Node *P = List.Hdr->Forward[0];
while (P != NIL) {

    /* examine P->Data here */

    P = P->Forward[0];
}
```

- *Space*. The amount of memory required to store a value should be minimized. This is especially true if many small nodes are to be allocated.

  - ♦ For hash tables, only one forward pointer per node is required. In addition, the hash table itself must be allocated.

  - ♦ For red-black trees, each node has a left, right, and parent pointer. In addition, the color of each node must be recorded. Although this requires only one bit, more space may be allocated to ensure that the size of the structure is properly aligned. Therefore each node in a red-black tree requires enough space for 3-4 pointers.

  - ♦ For skip lists, each node has a level-0 forward pointer. The probability of having a level-1 pointer is ½. The probability of having a level-2 pointer is ¼. In general, the number of forward pointers per node is

$$n = 1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2 .$$

- *Time*. The algorithm should be efficient. This is especially true if a large dataset is expected. Table 3-2 compares the search time for each algorithm. Note that worst-case behavior for hash tables and skip lists is extremely unlikely. Actual timing tests are described below.

- *Simplicity*. If the algorithm is short and easy to understand, fewer mistakes may be made. This not only makes *your* life easy, but the maintenance programmer entrusted with the task of making repairs will appreciate any efforts you make in this area. The number of statements required for each algorithm is listed in Table 3-2.

method	*statements*	*average time*	*worst-case time*
hash table	26	***O*** (1)	***O*** (n)
unbalanced tree	41	***O*** (lg n)	***O*** (n)
red-black tree	120	***O*** (lg n)	***O*** (lg n)
skip list	55	***O*** (lg n)	***O*** (n)

Table 3-2: Comparison of Dictionaries

Average time for insert, search, and delete operations on a database of 65,536 (2^{16}) randomly input items may be found in Table 3-3. For this test the hash table size was 10,009 and 16 index levels were allowed for the skip list. Although there is some variation in the timings for the four methods, they are close enough so that other considerations should come into play when selecting an algorithm.

method	*insert*	*search*	*delete*
hash table	18	8	10
unbalanced tree	37	17	26
red-black tree	40	16	37
skip list	48	31	35

Table 3-3: Average Time (μs), 65536 Items, Random Input

order	*count*	*hash table*	*unbalanced tree*	*red-black tree*	*skip list*
	16	4	3	2	5
random	256	3	4	4	9
input	4,096	3	7	6	12
	65,536	8	17	16	31
	16	3	4	2	4
ordered	256	3	47	4	7
input	4,096	3	1,033	6	11
	65,536	7	55,019	9	15

Table 3-4: Average Search Time (us)

Table 3-4 shows the average search time for two sets of data: a random set, where all values are unique, and an ordered set, where values are in ascending order. Ordered input creates a worst-case scenario for unbalanced tree algorithms, as the tree ends up being a simple linked list. The times shown are for a single search operation. If we were to search for all items in a database of 65,536 values, a red-black tree algorithm would take .6 seconds, while an unbalanced tree algorithm would take 1 hour.

4. Very Large Files

The previous algorithms have assumed that all data reside in memory. However, there may be times when the dataset is too large, and alternative methods are required. In this section, we will examine techniques for sorting (*external sorts*) and implementing dictionaries (*B-trees*) for very large files.

4.1 External Sorting

One method for sorting a file is to load the file into memory, sort the data in memory, then write the results. When the file cannot be loaded into memory due to resource limitations, an *external sort* applicable. We will implement an external sort using *replacement selection* to establish initial runs, followed by a *polyphase merge sort* to merge the runs into one sorted file. I highly recommend you consult Knuth [1998], as many details have been omitted.

Theory

For clarity, I'll assume that data is on one or more reels of magnetic tape. Figure 4-1 illustrates a 3-way polyphase merge. Initially, in phase A, all data is on tapes T1 and T2. Assume that the beginning of each tape is at the bottom of the frame. There are two sequential *runs* of data on T1: 4-8, and 6-7. Tape T2 has one run: 5-9. At phase B, we've merged the first run from tapes T1 (4-8) and T2 (5-9) into a longer run on tape T3 (4-5-8-9). Phase C is simply renames the tapes, so we may repeat the merge again. In phase D we repeat the merge, with the final output on tape T3.

phase	T1	T2	T3
A	7 6 8 4	 9 5	
B	 7 6		9 8 5 4
C	9 8 5 4	 7 6	
D			9 8 7 6 5 4

Figure 4-1: Merge Sort

Several interesting details have been omitted from the previous illustration. For example, how were the initial runs created? And, did you notice that they merged *perfectly*, with no extra runs on any tapes? Before I explain the method used for constructing initial runs, let me digress for a bit.

In 1202, Leonardo Fibonacci presented the following exercise in his *Liber Abbaci* (Book of the Abacus): "How many pairs of rabbits can be produced from a single pair in a year's time?" We may assume that each pair produces a new pair of offspring every month, each pair becomes fertile at the age of one month, and that rabbits never die. After one month, there will be 2 pairs of rabbits; after two months there will be 3; the following month the original pair and the pair born during the first month will both usher in a new pair, and there will be 5 in all; and so on. This series, where each number is the sum of the two preceding numbers, is known as the Fibonacci sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Curiously, the Fibonacci series has found widespread application to everything from the arrangement of flowers on plants to studying the efficiency of Euclid's algorithm. There's even a *Fibonacci Quarterly* journal. And, as you might suspect, the Fibonacci series has something to do with establishing initial runs for external sorts.

Recall that we initially had one run on tape T2, and 2 runs on tape T1. Note that the numbers {1,2} are two sequential numbers in the Fibonacci series. After our first merge, we had one run on T1 and one run on T2. Note that the numbers {1,1} are two sequential numbers in the Fibonacci series, only one *notch* down. We could predict, in fact, that if we had 13 runs on T2, and 21 runs on T1 {13,21}, we would be left with 8 runs on T1 and 13 runs on T3 {8,13} after one pass. Successive passes would result in run counts of {5,8}, {3,5}, {2,3}, {1,1}, and {0,1}, for a total of 7 passes. This arrangement is ideal, and will result in the minimum number of passes. Should data actually be on tape, this is a big savings, as tapes must be mounted and rewound for each pass. For more than 2 tapes, *higher-order* Fibonacci numbers are used.

Initially, all the data is on one tape. The tape is read, and runs are distributed to other tapes in the system. After the initial runs are created, they are merged as described above. One method we could use to create initial runs is to read a batch of records into memory, sort the records, and write them out. This process would continue until we had exhausted the input tape. An alternative algorithm, *replacement selection*, allows for longer runs. A buffer is allocated in memory to act as a holding place for several records. Initially, the buffer is filled. Then, the following steps are repeated until the input is exhausted:

- Select the record with the smallest key that is ≥ the key of the last record written.
- If all keys are smaller than the key of the last record written, then we have reached the end of a run. Select the record with the smallest key for the first record of the next run.
- Write the selected record.
- Replace the selected record with a new record from input.

Figure 4-2 illustrates replacement selection for a small file. The beginning of the file is to the right of each frame. To keep things simple, I've allocated a 2-record buffer. Typically, such a buffer would hold thousands of records. We load the buffer in step B, and write the record with the smallest key (6) in step C. This is replaced with the next record (key 8). We select the smallest key ≥ 6 in step D. This is key 7. After writing key 7, we replace it with key 4. This process repeats until step F, where our last key written was 8, and all keys are less than 8. At this point, we terminate the run, and start another.

Step	*Input*	*Buffer*	*Output*
A	5-3-4-8-6-7		
B	5-3-4-8	6-7	
C	5-3-4	8-7	6
D	5-3	8-4	7-6
E	5	3-4	8-7-6
F		5-4	3 | 8-7-6
G		5	4-3 | 8-7-6
H			5-4-3 | 8-7-6

Figure 4-2: Replacement Selection

This strategy simply utilizes an intermediate buffer to hold values until the appropriate time for output. Using random numbers as input, the average length of a run is twice the length of the buffer. However, if the data is somewhat ordered, runs can be extremely long. Thus, this method is more effective than doing partial sorts.

When selecting the next output record, we need to find the smallest key ≥ the last key written. One way to do this is to scan the entire list, searching for the appropriate key. However, when the buffer holds thousands of records, execution time becomes prohibitive. An alternative method is to use a binary tree structure, so that we only compare lg *n* items.

Implementation

Source for the external sort algorithm may be found in file **`ext.c`**. Function **`makeRuns`** calls **`readRec`** to read the next record. Function **`readRec`** employs the replacement selection algorithm (utilizing a binary tree) to fetch the next record, and **`makeRuns`** distributes the records in a Fibonacci distribution. If the number of runs is not a perfect Fibonacci number, *dummy* runs are simulated at the beginning of each file. Function **`mergeSort`** is then called to do a polyphase merge sort on the runs.

4.2 B-Trees

Dictionaries for very large files typically reside on secondary storage, such as a disk. The dictionary is implemented as an index to the actual file and contains the key and record address of data. To implement a dictionary we could use red-black trees, replacing pointers with offsets from the beginning of the index file, and use random access to reference nodes of the tree. However, every transition on a link would imply a disk access, and would be prohibitively

expensive. Recall that low-level disk I/O accesses disk by sectors (typically 256 bytes). We could equate node size to sector size, and group several keys together in each node to minimize the number of I/O operations. This is the principle behind B-trees. Good references for B-trees include Knuth [1998] and Cormen [1998]. For B^+-trees, consult Aho [1983].

Theory

Figure 4-3 illustrates a B-tree with 3 keys/node. Keys in internal nodes are surrounded by pointers, or record offsets, to keys that are less than or greater than, the key value. For example, all keys less than 22 are to the left and all keys greater than 22 are to the right. For simplicity, I have not shown the record address associated with each key.

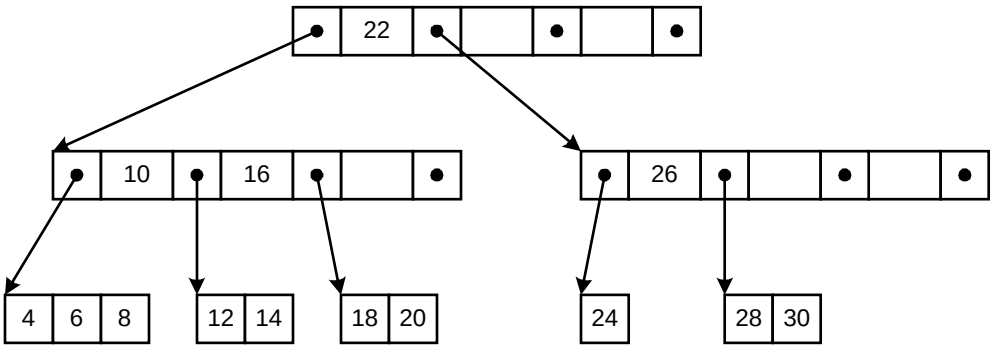

Figure 4-3: B-Tree

We can locate any key in this 2-level tree with three disk accesses. If we were to group 100 keys/node, we could search over 1,000,000 keys in only three reads. To ensure this property holds, we must maintain a balanced tree during insertion and deletion. During insertion, we examine the child node to verify that it is able to hold an additional node. If not, then a new sibling node is added to the tree, and the child's keys are redistributed to make room for the new node. When descending for insertion and the root is full, then the root is spilled to new children, and the level of the tree increases. A similar action is taken on deletion, where child nodes may be absorbed by the root. This technique for altering the height of the tree maintains a balanced tree.

	B-Tree	*B*-Tree*	*B^+-Tree*	*B^{++}-Tree*
data stored in	any node	any node	leaf only	leaf only
on insert, split	1 x 1 → 2 x 1/2	2 x 1 → 3 x 2/3	1 x 1 → 2 x 1/2	3 x 1 → 4 x 3/4
on delete, join	2 x 1/2 → 1 x 1	3 x 2/3 → 2 x 1	2 x 1/2 → 1 x 1	3 x 1/2 → 2 x 3/4

Table 4-1: B-Tree Implementations

Several variants of the B-tree are listed in Table 4-1. The *standard* B-tree stores keys and data in both internal and leaf nodes. When descending the tree during insertion, a full child node is first redistributed to adjacent nodes. If the adjacent nodes are also full, then a new node is created, and ½ the keys in the child are moved to the newly created node. During deletion, children that are ½ full first attempt to obtain keys from adjacent nodes. If the adjacent nodes are also ½ full, then two nodes are joined to form one full node. B*-trees are similar, only the nodes

are kept 2/3 full. This results in better utilization of space in the tree, and slightly better performance.

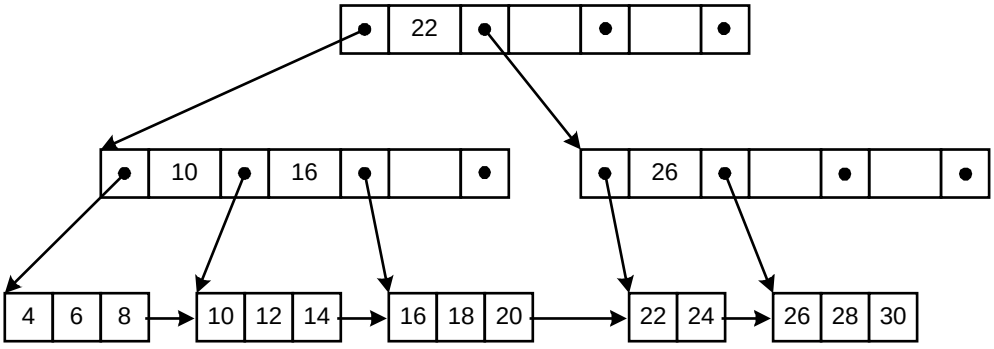

Figure 4-4: B-$^{+}$Tree

Figure 4-4 illustrates a B^{+}-tree. *All* keys are stored at the leaf level, with their associated data values. Duplicates of the keys appear in internal parent nodes to guide the search. Pointers have a slightly different meaning than in conventional B-trees. The left pointer designates all keys less than the value, while the right pointer designates all keys *greater than or equal to* (GE) the value. For example, all keys less than 22 are on the left pointer, and all keys greater than or equal to 22 are on the right. Notice that key 22 is duplicated in the leaf, where the associated data may be found. During insertion and deletion, care must be taken to properly update parent nodes. When modifying the first key in a leaf, the last GE pointer found while descending the tree will require modification to reflect the new key value. Since all keys are in leaf nodes, we may link them for sequential access.

The last method, B^{++}-trees, is something of my own invention. The organization is similar to B^{+}-trees, except for the split/join strategy. Assume each node can hold *k* keys, and the root node holds 3*k* keys. Before we descend to a child node during insertion, we check to see if it is full. If it is, the keys in the child node and two nodes adjacent to the child are all merged and redistributed. If the two adjacent nodes are also full, then another node is added, resulting in four nodes, each ¾ full. Before we descend to a child node during deletion, we check to see if it is ½ full. If it is, the keys in the child node and two nodes adjacent to the child are all merged and redistributed. If the two adjacent nodes are also ½ full, then they are merged into two nodes, each ¾ full. Note that in each case, the resulting nodes are ¾ full. This is halfway between ½ full and completely full, allowing for an equal number of insertions or deletions in the future.

Recall that the root node holds 3*k* keys. If the root is full during insertion, we distribute the keys to four new nodes, each ¾ full. This increases the height of the tree. During deletion, we inspect the child nodes. If there are only three child nodes, and they are all ½ full, they are gathered into the root, and the height of the tree decreases.

Another way of expressing the operation is to say we are *gathering* three nodes, and then *scattering* them. In the case of insertion, where we need an extra node, we scatter to four nodes. For deletion, where a node must be deleted, we scatter to two nodes. The symmetry of the operation allows the gather/scatter routines to be shared by insertion and deletion in the implementation.

Implementation

Source for the B^{++}-tree algorithm may be found in file **`btr.c`**. In the implementation-dependent section, you'll need to define **`bAdrType`** and **`eAdrType`**, the types associated with B-tree file offsets and data file offsets, respectively. You'll also need to provide a callback function that is used by the B^{++}-tree algorithm to compare keys. Functions are provided to insert/delete keys, find keys, and access keys sequentially. Function **`main`**, at the bottom of the file, provides a simple illustration for insertion.

The code provided allows for multiple indices to the same data. This was implemented by returning a *handle* when the index is opened. Subsequent accesses are done using the supplied handle. Duplicate keys are allowed. Within one index, all keys must be the same length. A binary search was implemented to search each node. A flexible buffering scheme allows nodes to be retained in memory until the space is needed. If you expect access to be somewhat ordered, increasing the **`bufCt`** will reduce paging.

5. Bibliography

Aho, Alfred V. and Jeffrey D. Ullman [1983]. Data Structures and Algorithms. Addison-Wesley, Reading, Massachusetts.

Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest [1990]. Introduction to Algorithms. McGraw-Hill, New York.

Knuth, Donald. E. [1998]. The Art of Computer Programming, Volume 3, Sorting and Searching. Addison-Wesley, Reading, Massachusetts.

Pearson, Peter K [1990]. Fast Hashing of Variable-Length Text Strings. *Communications of the ACM*, 33(6):677-680, June 1990.

Pugh, William [1990]. Skip lists: A Probabilistic Alternative To Balanced Trees. *Communications of the ACM*, 33(6):668-676, June 1990.