Section 8.1

1) Let us assume that a search does not start at the beginning of a list when the Sequential Search algorithm (Algorithm 1.1) is used, but rather starts wherever the list index was left at the termination of the preceding search. Let us further assume that the item for which we are searching is selected randomly and independently of the destinations of previous searches. Under these assumptions, what would be the average number of comparisons?

Since the item is selected randomly, it has equal probabilities to be in any position. If it's not found after the current element, the search will continue at the beginning of the array, but on the average n/2 elements will have to be searched, as in Algorithm 1.1.

2) Problem: Find all of the comment elements of two arrays S and T of sizes m and n respectively.

Inputs: positive integers m and n, arrays S[] and T[] indexed from 1 to m and 1 to n, respectively.

Outputs: Array U[] containing all of the common elements of S and T.

```
keytype[] common(int m, int n, keytype S[], keytype T[]) {
        index  i, j=1;
        keytype H[1…2*m];
        keytype U[1…min(n,m)];
        for(i=1; i<m; i++) {
                int hash = hashCode(S[i]) % (2*m);
                while(H[hash] != null)
                        hash++;
                H[hash] = S[i];
        }
        for(i=1; i<n; i++) {
                int hash = hashCode(T[i]) % (2*m);
                while(H[hash] != null) {
                        if(H[hash]==T[i]) {
                                U[j] = T[i];
                                j++;
                                break;
                }
                hash++;
            } //end while
        } //end for
        return U;
}
```

The algorithm places the elements of the first array into a hash table using linear probing. This takes Θ(m) time in the average case for a suitable load factor (<0.5 is used in the algorithm). Then each element in the second array is searched for in the hash table. This takes Θ(n) time again in the average case for a reasonable size hash table. The total runtime is thus Θ(m+n). It is assumed that a suitable, constant-time hash function hashCode() is available to generate a hash code for an element appropriate to the element's keytype.

3) Improve the Binary Search algorithm (Algorithm 1.5) assuming a successful search. Analyze your algorithm and show the results using order notation.

If we know in advance that the searched key will be found in the array, the test **`low <= high`** does not need to be performed, because by the time **`low`** becomes greater than **`mid`**, the key has been found.

```
void binsearch (int n, const keytype S[] , keytype x, index& location) {
        index low , high , mid;
        low = 1; high = n;
        location = 0;
        while (~~low <= high &&~~ location = = 0) {      //test skipped
            mid = floor((low+high)/2);
            if (x  = =  S[mid])
                location = mid;
            else if (x < S[mid])
                high = mid − 1;
            else
                low = mid + 1;
        }
}
```

Since the complexity is derived in the text by counting only comparisons that involve array elements, this change does not impact it at all.

4) Show that, if x is in the array and is equally probable to be in each of the array slots, the average-case time complexity for Binary Search (Algorithm 1.5) is bounded approximately by

$$\lfloor \lg n \rfloor - 1 \le A(n) \le \lfloor \lg n \rfloor .$$

Let k-1 be the deepest complete level in the binary tree. From Lemma 3, we know that the tree is nearly complete, so there can be only one more level after that: k. The number of nodes down to level k-1 is 2^k-1, and, following the hint, the number of nodes at the bottom level, k, is n – (2^k – 1).

The contributions to the TND are as follows:

- As in the proof on p.344, for the complete tree (down to level k-1) we have (k-1)2^k + 1.
- The bottom level is reached through k + 1 nodes starting from the root, so we have (k+1)(n – (2^k – 1)).

Adding up the two terms above, we obtain the TND for the whole tree, and dividing by n we obtain the average:

$$A(n) = TND/n = k + 1 - \frac{2^{k+1}}{n} + \frac{k+2}{n}$$

Since the last term tends to 0 when n tends to infinity, we can neglect it:

$$A(n) \approx k + 1 - \frac{2^{k+1}}{n} \qquad (*)$$

Since the bottom level can have from 0 to 2^k – 1 nodes, the total number of nodes in the tree n is between 2^k – 1 and 2(2^k – 1), so we have the double inequality

$2^k - 1 \le n \le 2^{k+1} - 1$, which can be processed in two ways to apply to (*):

- Divide by n to obtain approximately $2^k/n \le 1 \le 2^{k+1}/n$ (**)
- Take $\log_2$ to obtain approximately $k \le \lg n < k + 1$, which shows that

$$\lfloor \lg n \rfloor = k \qquad (***)$$

By using (***) and right-hand-side of (**) in (*), we obtain

$A(n) \le \lfloor \lg n \rfloor$ (α)

Similarly, using (***) and left-hand-side of (**) in (*), we obtain

$\lfloor \lg n \rfloor - 1 \le A(n)$ (β)

(α) and (β) are the two sides of the result we needed to prove.

5) Suppose that all of the following 2n+1 possibilities are equally probable:

$$\begin{array}{ll} x = s_i & \text{for some } i \text{ such that } 1 \le i \le n, \\ x < s_1, & \\ s_i < x < s_{i+1} & \text{for some } i \text{ such that } 1 \le i \le n-1, \\ x > s_n. & \end{array}$$

Show that the average-case time complexity of the Binary Search algorithm (Algorithm 1.5) is bounded approximately by

$$\lfloor \lg n \rfloor - \frac{1}{2} \le A(n) \le \lfloor \lg n \rfloor + \frac{1}{2}.$$

Let k-1 be the deepest complete level in the binary tree. From Lemma 3, we know that the tree is nearly complete, so there can be only one more level after that: k. The number of nodes down to level k-1 is 2^k-1, and, following the hint, the number of nodes at the bottom level, k, is n – (2^k – 1). It is easy to see that the position of the nodes on the bottom level does not change the number of successful or unsuccessful searches on either level k-1 or k, so we can assume, for instance, an *essentially complete* binary tree (Fig. 7.4).

The contributions to the TND are as follows:

- For the complete tree (down to level k-1) we have $(k-1)2^k + 1 \;\; + \;\; k(n + 1 – [n – 2^k + 1])$. For clarity, the square brackets above include the subtracted term that is not in the proof for the complete case on p.345.
- The bottom level is reached through k + 1 nodes starting from the root, so we have $(k+1)[n – 2^k + 1])$.

Adding up the two terms above, we obtain the TND for the whole tree, and dividing by 2n+1 we obtain the average:

$$A(n) = TND/(2n+1) = \frac{(k+1)n + k2^k - 2^{k+1}}{2n+1} + \frac{k+2}{2n+1}$$

Since the last term tends to 0 when n tends to infinity, we can neglect it. Then divide both numerator and denominator by n and neglect the vanishing terms. We have:

$$A(n) \approx \frac{k+1}{2} + \frac{k2^k}{2n} - \frac{2^k}{n} \qquad (*)$$

Since the bottom level can have from 0 to 2^k – 1 nodes, the total number of nodes in the tree n is between 2^k – 1 and 2^{k+1} – 1, so we have the double inequality

$2^k – 1 \le n \le 2^{k+1} – 1$, which can be processed in two ways to apply to (*):

- Divide by n to obtain approximately $2^k/n \le 1 \le 2^{k+1}/n$ (**)
- Take $\log_2$ to obtain approximately $k \le \lg n < k + 1$, which shows that

$$\lfloor \lg n \rfloor = k \qquad (***)$$

By using (***) in (*), we obtain

$$A(n) \approx \frac{\lfloor \lg n \rfloor + 1}{2} + \frac{\lfloor \lg n \rfloor}{2} \cdot \frac{2^k}{n} - \frac{2^k}{n}$$

and now application of (***) yields the result we needed to prove.

6) Complete the proof of Lemma 8.6.

To prove the Lemma, it is left to show that $\sum_{i=1}^{n} c_i \geq$ minTND(n), where c_i is the number of nodes in the shortest path to a node that compares the search key x to the array item s_i. The reason we say "shortest path" is that there may be other comparisons of x to s_i in the tree, located at a larger depth, therefore when we add the length of the paths to those nodes, we find that $\sum_{i=1}^{n} c_i \geq$ TND(n), which, in turn, must be by definition ≥ minTND(n), and the inequality follows by transitivity.

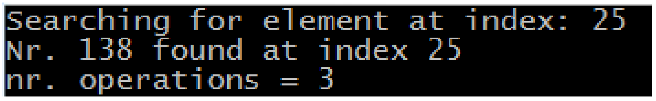
Section 8.2

7) Implement the Binary Search, Interpolation Search, and Robust Interpolation Search algorithms on your system and study their best-case, average-case, and worst-case performances using several problem instances.

The following is an implementation of Interpolation Search:

```
#include <iostream>
#include <random>
#include <ctime>
using namespace std;

#define N 30
int S[N] = { -420, -40, -38, -37, -36, -35, -34, -30, -29, -28,
              -10, -5,    1,   2,   3,   4,   5,   6, 100, 102,
              110, 120, 130, 135, 136, 138, 140, 143, 150, 151};
int counter = 0;

int interpsrch(int n, int S[], int x) {
	int low, mid, high;
	low = 0; high = n-1;
	while (low <= high) {
		counter++;
		mid = low + floor(
			(high-low) * (x-S[low]) / (float)(S[high]-S[low])
```

```
                    );
            if (x == S[mid])
                    return mid;
            if (x < S[mid])
                    high = mid - 1;
            else
                    low = mid + 1;
        }
        return -1; //if x not found in array
}

int main() {
        srand(time(0));             //seed the random number generator
        int i = rand() % N;         //generate random index to search

        cout <<"Searching for element at index: " <<i <<endl;
        cout <<"Nr. " <<S[i] <<" found at index "
               <<interpsrch(N, S, S[i]) <<endl;
        cout <<"nr. operations = " <<counter <<endl;

        return 0;
}
```



```
Searching for element at index: 25
Nr. 138 found at index 25
nr. operations = 3
```

8) Show that the average-case time complexity of Interpolation Search is in Θ(lg(lgn)), assuming the keys are uniformly distributed and that search key x is equally probable to be in each of the array slots.

Solutions will vary.

9) Show that the worst-case time complexity of Interpolation Search is in Θ($(\lg n)^2$), assuming the keys are uniformly distributed and that search key x is equally probable to be in each of the array slots.

Solutions will vary.

Section 8.3

10) Problem: Find the largest key in a binary search tree.

Inputs: Reference to a node n, representing the root of the tree.

Outputs: an integer m representing the maximum key of the tree.

```
int maxKey(const node& n) {
        if(n.right == null) return n.key;
        return maxKey(n.right);
}
```

The algorithm traverses the rightmost branch of the tree. The every-case complexity is O(h), where h is the height of the tree. If there are no restrictions on the balance of the tree, this could have a worst-case complexity of O(n), where n is the number of nodes in the tree. If the tree is restricted to be balanced, then the every-case complexity is Θ(log n).

11) Theorem 8.3 states that, for a successful search, the average search time over all inputs containing n keys, using binary search trees, is in Θ(lg n). Show that this result still holds if we consider an unsuccessful search as well.

If key k is in the root, there are still k-1 keys in the left subtree and n-k in the right subtree, but now we also have k of the intervals for unsuccessful search on the left and n-k+1 on the right; the total cases on the left is 2k-1, and on the right 2n-2k+1.

The derivations now closely parallel those in Theorem 8.3, but with a slightly different recursion. The conditional average is

$$A(n|k) = A(k-1)\frac{2k-1}{2n+1} + A(n-k)\ \frac{2n-2k+1}{2n+1} + 1$$

And therefore the entire average is

$$A(n) = \frac{1}{n}\sum_{k=1}^{n}\ \left(A(k-1)\frac{2k-1}{2n+1} + A(n-k)\ \frac{2n-2k+1}{2n+1} + 1 \right)$$

We denote A(m)·(2m+1) by C(m), and, after manipulations, obtain

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{4n-1}{n(n+1)}$$

We denote C(n)/(n+1) by a_n, and write $\frac{4n-1}{n(n+1)} = \frac{5}{n+1} - \frac{1}{n}$ to obtain the recursion

$$a_n = a_{n-1} + \frac{5}{n+1} - \frac{1}{n}$$

We “unroll” the recurrence by successive substitutions and use the logarithm formula from Example A.9 to get the closed-form expression $a_n \approx 4\ln n - 5$, which means $C(n) \approx 2.76(n+1)\lg n$, and $A(n) \approx 1.38\lg n$.

12) void delete(node root&, keytype k) {

```
        node current = root;
        node parent = root;
        boolean leftChild = true;
//search for node to delete
while(current.key != k) {
                parent = current;
                if(k < current.key) {
                        leftChild = true;
                        current = current.left;
}
else {
        leftChild = false;
        current = current.right;
}
if(current == null) return; //return if not found
}
//node to delete has no children
if(current.left == null && current.right == null) {
        if(current == root) root = null;
        else if(leftChild) parent.left = null;
        else parent.right = null;
}
//node to delete has only a left child
else if(current.right == null) {
        if(current == root) root = current.left;
        else if(leftChild) parent.left = current.left;
        else parent.right = current.left;
}
//node to delete has only a right child
else if(current.left == null) {
        if(current == root) root = current.right;
        else if(leftChild) parent.left = current.right;
        else parent.right = current.right;
}
//node to delete has two children
else {
        node successor = getSuccessor(current);
        if(current == root) root = successor;
        else if(leftChild) parent.left = successor;
        else parent.right = successor;
        successor.left = current.left;
}
return true;
}

node getSuccessor(node n&) {
        node sParent = n;
        node s = n;
        node current = n.right;
        while(current != null) {
                sParent = s;
                s = current;
                current = current.left;
}
if(s != n.right) {
        sParent.left = s.right;
        s.right = n.right;
}
return s;
}
```

The algorithm considers the four possible cases when deleting a node: the node has no children, the node has only a left child, the node has only a right child, and the node has two children. The algorithm traverses at most one branch of the tree for any given key. If the tree is balanced, the complexity is thus O(log n). If there are no balance restrictions the complexity can degenerate to O(n) in the worst case.

13) Write an algorithm that creates a 3–2 tree from a list of keys. Analyze your algorithm and show the results using order notation.

The keys from the list L(1..n) are inserted successively in the tree T, starting with an empty tree:

```
int main() {
    T = root containing L(1);                              //insert first key
    Insert L(2) in root, either left or right of L(1)      //insert second key
    Insert L(3) in root, in one of 3 positions relative to L(1) and L(2);//start inserting third
    key
                                                           //This is not a valid 2-3 tree!
    Leave only middle key in root;
    Create 2 children with left and right keys;            //Finished inserting third key
    for (i = 4; i <= n; i++) {
        w = find(T, L(i));
        recursive_insert(T, w, L(i));
    return 0;
}

void general_insert(T, w, x) {
    if (w has 2 children) {                                //w contains only one key
        Insert x as either w.left or w.right;
        Insert k as the appropriate child of w;
    }
    else if (w has 3 children) {                           //w is full: it contains two keys
        Make x the appropriate new child of w;             //w has 4 children!
        Create a new internal node v.
        Give w's two rightmost children to v;
        Place either child's key in v. Let this key be z;
        recursive_insert(T, w's parent, z);
}
```

The function find(T, x) navigates the tree T from the root to the internal node under which the key x should be inserted.

In the worst case, the insertion of a key requires moving down lg n levels, then, recursively, up lg n levels, so the complexity is O(nlg n).

14) void inorder(node n) {

```
        if(n.left != null) inorder(n.left);
        print(n.leftKey);
        if(n.mid != null) inorder(node.mid);
if(n.rightKey != null) print(n.rightKey);
if(n.right != null) inorder(n.right);
}
}
```

Assumes each node contains two key variables, *leftKey* and *rightKey*, and three child variables, *left*, *mid*, and *right*. If a node has only 1 key, then *rightKey* and *mid* are null.

15) If the three elements with the smallest hash values are inserted in the order 567-21-7100, 267-89-5401, 234-76-8100, then:

Element 567-21-7100 is placed in bucket 0.

Element 267-89-5401 is placed in bucket 1.

Element 234-76-8100 is placed in bucket 2.

Element 330-38-0172 is placed in bucket 72.

Element 435-87-9872 is placed in bucket 73.

Element 897-38-6872 is placed in bucket 74.

Element 432-87-0199 is placed in bucket 99.

16) Open hashing incurs the overhead of maintaining a linked list structure at each non-empty spot in the hash array and traversing it each time a search is performed. However, its performance is not as seriously dependent on load factor as linear probing, and thus a smaller hash array size can generally be used than with linear probing, saving space. Linear probing is easier to implement since there is no overhead of maintaining lists, but a high load factor can seriously degrade performance, and so care must be taken to have a large enough hash array size, which takes up more memory.

17) void delete(keytype H[], keytype key) {

```
    int hash = hashCode(key);
    while(H[hash] != null) {
        if(H[hash]== key) {
            H[hash] = *special value*;
            return;
        }
        hash++;
    }
}
```

The algorithm takes the hash array in which the element resides and searches for the element in the array using linear probing. The hashCode() function is the same hash function that was used to insert the element into the array. If the element it found, the position in the array in which it is found is set to a special value that is known not to be a possible key. For example, if the array contains only non-negative integers, this value could be -1. In this way, the search function would have to be modified to skip over (but not terminate on) this value. Additionally, the insert function would insert a new element into a slot that was either null or contained this special value. Doing this, insertion, search, and deletion all take O(1) time assuming suitable load factors and hash functions as usual.

18) Let the second hash function be s(key) = (key/10000)%100. Assuming that the three elements with the smallest hash values are inserted in the order 567-21-7100, 267-89-5401, 234-76-8100, the elements would be mapped as follows:

Element 567-21-7100 is placed in bucket 0.

Element 267-89-5401 is placed in bucket 1.

Element 234-76-8100 is placed in bucket 76.

Element 330-38-0172 is placed in bucket 72.

Element 435-87-9872 is placed in bucket 59.

Element 897-38-6872 is placed in bucket 10.

Element 432-87-0199 is placed in bucket 99.

Section 8.5

19) Modify Algorithm 8.4 (Find Smallest and Largest Keys by Pairing Keys) so that it works when n (the number of keys in the given array) is odd and show that its time complexity is given by 3n/2 −2 if n is even and 3n/2 – 3/2 if n is odd.

The only change for n odd is in the if statement before the for loop. Instead of

```
if (S[1] < S[2]) {
        small = S[1];
        large = S[2];
}
else {
        small = S[2];
        large = S[1];
}
```

we "consume" only one array element, so the remainder is left even for the for loop:

```
small = S[1];
large  = S[1];
```

For even n we have 1 + 3(n/2 – 1) = 3n/2 – 2 comparisons.

For odd n, we have 0 + 3(n-1)/2 = 3n/2 – 3/2 comparisons.

20) Complete the proof of Theorem 8.8. That is, show that a deterministic algorithm that finds the smallest and largest of n keys only by comparisons of keys must in the worst case do at least (3n−3)/2 comparisons if n is odd.

If n is odd, let n = 2k + 1. The algorithm can make at most k = (n – 1)/2 "two unit" comparisons, which gain 2k = n – 1 units of information. This leaves (2n – 2) – (n – 1) = n – 1 units to be gained from "one unit" comparisons, which requires n – 1 comparisons. The total nr. of comparisons is (n – 1)/2 + n – 1 = (3n – 3)/2.

21) Write an algorithm for the method discussed in Section 8.5.3 for finding the second-largest key in a given array.

We need two specialized structures for this algorithm:

```
struct list_element {
        struct array_element * array_ptr;
        struct list_element * list_ptr;
};
struct array_element {
        int data;
        bool loser;      //true if node lost
        struct list_element * list_ptr;
};
```

The array is stored in

```
struct array_element a[1..n];
```

whose loser fields are initially all set to false, and pointers are initially all set to NULL.

The tournament rounds are implemented with a for loop:

```
for (int i = 0; i < lg n; i++){
        int j = 0;
        while (j < n){
                choose next non-loser pair in a;
                compare pair;
                update loser field of the loser;
                append element in winner's list with array+ptr = &loser;
        }
}
```

After the for loop, we find the sole winner (non-loser) left, and perform linear search in its list.

22) Show that for n in general, the total number of comparisons needed by the method discussed in Section 8.5.3 for finding the second-largest key in a given array is

$T(n) = n + \lceil \lg n \rceil - 2.$

As indicated in the text, when $n = 2^k + m$ is not a power of 2, we pad the array with $2^k - m$ "minus infinity" elements in order for the tournament scheme to work, however we don't need to physically compare any of the pad elements – the algorithm knows by their index if they're pads or actual elements. When two pads need to be compared, any of them (say, the first) can be declared the winner, and when a pad is compared with a real element, the real element is the winner. This way, the complexity of the tournament part is still n – 1 comparisons.

The nr. of levels in the tournament tree is $\lceil \lg n \rceil$, so their sequential search requires $\lceil \lg n \rceil$ - 1 comparisons.

The total is $T(n) = n + \lceil \lg n \rceil - 2$.

23) int medianOf5(int a[]) {
```
    if(a[1] > a[2])   swap(a, 1, 2);
    if(a[4] > a[5])   swap(a, 4, 5);
    if(a[1] > a[4])   swap(a, 1, 4);
    if(a[3] > a[2]) {
    if(a[2] < a[4])   return min(a[3], a[4]);
        else   return min(a[2], a[5]);
    }
    else {
        if(a[3] > a[4])   return min(a[3], a[5]);
        else   return min(a[2], a[4]);
    }
    }
    void swap(int a[], int x, int y) {
        int temp = a[x];
        a[x] = a[y];
        a[y] = temp;
}
```
Note that each call to min() counts as one comparison. Thus, each possible control execution path has exactly 6 comparisons.

24) Use induction to show that the worst-case time complexity of Algorithm 8.6

(Selection Using the Median) is bounded approximately as $W(n) \leq 22n$.

We use the (approximate) recursion formula on p.384: $W(n) \approx W(7n/10) + W(n/5) + 11n/5$, valid for all positive integers n.

The base case is already provided in the text: $W(n) \leq 22n$ is true for n up to 5. ($T(5) = 6$ comparisons).

Induction hypothesis: $W(k) \leq 22k$ is true for all k up to n.

Induction step: We prove that $W(k) \leq 22k$ is true for n + 1. Indeed, $W(n+1) \approx W(7(n+1)/10) + W((n+1)/5) + 11(n+1)/5$, and, since $7(n+1)/10 < n$, and $(n+1)/5 < n$, we can apply the hypothesis:

$W(n+1) \leq 22\cdot7(n+1)/10 + 22\cdot(n+1)/5 + 11(n+1)/5 = 22n + 22 \leq 22n$. Q.e.d.

25) Show that for an arbitrary m (group size), the recurrence for the worst-case time complexity of Algorithm 8.6 (Selection Using the Median) is given by

$$W(n) \approx W\left(\frac{(3m-1)n}{4}\right) + W\left(\frac{n}{m}\right) + an,$$

where a is a constant. This is Recurrence 8.2 in Section 8.5.4.

If we use groups of m instead of 5, the nr. of groups is n/m, and all but one of them have elements that could lie on either side of the median: n/m – 1 groups. In each of them, there are $\lfloor m/2 \rfloor$ such elements, so instead of $2(n/5 - 1)$ we have $\lfloor m/2 \rfloor \cdot (n/m - 1)$ elements that could lie on either side of the median (see p.381 of text).

The maximum nr. of elements on one side of the median of medians is

$$\frac{1}{2}(n-1-\left\lfloor\frac{m}{2}\right\rfloor(\frac{n}{m}-1)) + \left\lfloor\frac{m}{2}\right\rfloor\left(\frac{n}{m}-1\right) = \frac{1}{2}(n-1+\left\lfloor\frac{m}{2}\right\rfloor(\frac{n}{m}-1)) \leq \frac{1}{2}(n-1+\left(\frac{m}{2}\right)(\frac{n}{m}-1)) = \frac{3n-m}{4} - \frac{1}{2}.$$

As done in the text analysis, we drop the constant term, and keep only $\frac{3n-m}{4}$.

The number of comparisons required when calling *selection2* from *partition2* is n/m.

The number of comparisons needed to find the medians in the groups of m and to partition the array is linear in n.

Conclusion: $W(n) \approx W(\frac{3n-m}{4}) + W(\frac{n}{m}) + an$.

Note: The text formula for Recurrence 8.2 is wrong; it should be changed to the one above.

26) Use induction to show that W (n) ∈ Ω(nlg n) for the following recurrence.

This is Recurrence 8.2 in Section 8.5.4 where m (group size) is 3.

$$W(n) = W\left(\frac{2n}{3}\right) + W\left(\frac{n}{3}\right) + \frac{5n}{3}$$

For simplicity, we use $\log_3$ here, knowing that changing the base to 2 means only a constant multiplicative factor.

Base case: For n = 2 we have W(2) = W(1) + W(1) + 10/3 = 0 + 0 + 10/3 = 10/3, and nlg n= 2, so the maximum C that would satisfy 10/3 = W(2) ≥ $C2\log_3 2$ is C_{max} = 5/3 ≈ 2.64. (*)

Induction hypothesis: W(k) ≥ Cklg k, for k from 2 to n.

Induction step: We prove that W(n+1) ≥ C(n+1)lg(n+1):

W(n+1) = $W\left(\frac{2(n+1)}{3}\right)+W\left(\frac{n+1}{3}\right)+\frac{5(n+1)}{3}$, and, because each of the arguments of W on the right-hand-side are ≤ n, we can apply the hypothesis:

W(n+1) ≥ $C\frac{2n+2}{3}\log_3\frac{2n+2}{3}$ + $C\frac{n+1}{3}\log_3\frac{n+1}{3}+\frac{5n+5}{3}$, which after simple manipulations becomes $C(n+1)\log_3(n+1)$ + C(n+1)[$\frac{2\log_3 2}{3}-1+\frac{5}{3C}$], and the sum in the square brackets is negative for C ≥ 0.77 (**), making W(n+1) ≥ $C\frac{2n+2}{3}\log_3\frac{2n+2}{3}$.

From the inequalities (*) and (**) above, we conclude that the induction works with any C in the interval [0.77 … 2.64].

27) Show that the constant c in the inequality E(n,k) ≤ cn in the expected-value time complexity analysis of Algorithm 8.7 (Probabilistic Selection) cannot be less than 4.

We redo all the derivations following formula (8.3) on p.389 with an unknown constant C instead of 4. We obtain E(n, k) < C·3n/4 + n – 1. In order for the induction step to work, we need this to be less than C·n, which resolves to C > 4 – 1/n, and since 1/n tends to zero, this means C ≥

28) Implement Algorithms 8.5, 8.6, and 8.7 (Selection algorithms for finding the kth-smallest key in an array) on your system and study their best-case, average-case, and worst-case performances using several problem instances.

Implementations and performances will vary.

29) Write a probabilistic algorithm that determines whether an array of n elements has a majority element (the element that appears the most). Analyze your algorithm and show the results using order notation.

One solution is based on the observation that, if the majority element exists, it is also the median (if n = 2k+1 is odd), or immediately to the right of the median (k+1, if n = 2k is even). For simplicity, we assume n odd here: n = 2k+1, and we need the k+1-smallest key. We then apply the probabilistic algorithm for the median (Algorithm 8.7) to find the median M, and, once we've found it, we make one more pass through the array to count the number of occurrences of M. If that number is ≥ k+1, M is the majority element; if not, there is no majority element.

The complexity is ≈ 4n + n = 5n, therefore linear, Θ(n).

The simplest solution, however, is a "flow counting" algorithm invented in 1980 by Boyer and Moore. (It can be found in R. S. Boyer and J. S. Moore, **MJRTY: A fast majority vote algorithm**, in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, 1991.)

```
//Boyer-Moore majority algorithm
int MJRTY(int* S, int n) {
        int count_major = 0;
        int major;
         for (int i = 0; i < size; i++) {
                if (count_major == 0)
                        major = arr[i];
                if (arr[i] == major)
                        count_major ++;
                else
                        count_major --;
         } //end for (*)
         count_major = 0;
         for (i = 0; i < n; i++)
                if (arr[i] == major)
```

```
            count_major ++;
        if (count_major > n/2)
            return major;
        return -1;    //there's no majority element
}
```

Note: If we know in advance that there is a majority element, the algorithm can stop at point (*), because *major* contains that majority element. If, however, there is no majority element, at point (*) major contains a spurious, "late majority" element of S, as can be seen in this example:

S = {1, 1, 1, 2, 2, 2, 3} – here the three 1s and three 2s cancel each other and 3 gets stored in *major*.

In general, then, we need a second pass through the array to make sure that a majority element exists; this is done with the second for loop.

The every-time complexity is T(n) = 2n, so we don't need a probabilistic version of it.

Additional Exercises

30) Suppose a very large sorted list is stored in external storage. Assuming that this list cannot be brought into internal memory, develop a searching algorithm that looks for a key in this list. What major factor(s) should be considered when an external search algorithm is developed? Define the major factor(s), analyze your algorithm, and show the results using order notation.

Since accessing the external storage is about a million times slower than accessing the RAM, we want to minimize the number of external accesses (only reads if our problem is just searching). The complexity in this case is determined by counting only the external read operations.

- If all we know about the data is that it is sorted, the best algorithm is binary search (BS), with deterministic complexity lg n.
- If the data has even a modest degree of uniformity in distribution, interpolation search (IS) should be tried; although the index calculations are more complex than for BS, those calculations are very fast, since they are performed entirely in the RAM. True, the worst-case for IS is linear, but the average case is Θ(lg lg n), which is a great improvement.

Other factors are the particular mechanism used to transfer data from external storage to RAM, and the size of the keys:

All external storage devices (e.g. hard-disk drives – HDD) have a minimum block of information (e.g. 4KB or 16 KB) that is transferred in one access. If the keys are relatively small compared to the block size, then many of them can fit in one block, and the access speed can be improved by using a *sparse index*.

- The first key in each block is stored in RAM, with a pointer to the physical external location; when a key k is searched, its block is found through a fast RAM search, and then the block is read in 1 operation, Θ(1). Let m be the number of keys stored in one block. If n/m records can fit in the RAM, this method works.
- If n/m keys are still too many for the RAM, a *hierarchical index* can be used: We build a B-tree of the keys (or just the first key in each block, as described above), but only the first few levels of the tree are stored in the RAM – the subtrees under the deepest RAM level are stored in external storage. When a key k is searched, its subtree is first determined through a fast RAM search, and then we perform a read to bring that subtree in the RAM. Now we search the subtree, and another read will bring the key in the RAM. The total of read operations is 2, still Θ(1).

In real-life applications, the keys are only one part of a larger record, and the entire record must be retrieved, not just the key. If the rest of the record is large compared with the key, we can again use an *index* that stores only the keys, with pointers to the physical location of the record in external storage. Again, the index can be sparse, it can use B-trees, or hashing.

31) Discuss the advantages of using each of the following instead of the other:

(a) A binary search tree with a balancing mechanism

(b) A 3–2 tree

For equal number of keys n, the height of a balanced BST is $\approx \log_2 n$, whereas the height of a 3-2 tree is $\approx \log_3 n$. Since $\log_2 n / \log_3 n \approx 1.44$, the BST is about 44% deeper, resulting in 44% longer search time on average if the time to access a node dominates (e.g. nodes are in external storage). In practice, the branching factor of B-trees is a lot higher (100-99 trees are not uncommon), leading to much smaller depth than BSTs: $\log_2 n / \log_{100} n \approx 6.64$.

Because 3 keys can fit inside a node instead of 1, 3-2 trees need to be rebalanced less often than BSTs.

Consider, however, this example where both trees are entirely in the RAM:

We have $n = 2^{20} \approx 1$ million keys, and each key is stored in one word of memory. Pointers are also one word long.

- From the analysis of Algorithm 8.1, the average BST search examines 20 nodes, with 3 words per node (one key and 2 pointers), for a total of 60 words.
- The algorithm and analysis of B-trees are not presented in our text, but it is known that a 3-2 tree is on the average 75% full, so if it stores n actual keys, there is room in its nodes for $N = 2^{22}/3$ keys. The average search examines $\log_3 N = \log_3(2^{22}/3)$ nodes, with 5 words per node (2 keys and 3 pointers) for a total of $5 \cdot 22 \cdot \log_3(2) - 5 = 64.4$ words.

If the operations counted for complexity are word accesses in the RAM, the 3-2 tree has more complexity.

32) 1. Any situation in which the records need to be accessed in sorted order according to key: an example would be a student roster in which records are hashed by student last name that must be printed out in alphabetical order.

2. Any situation in which the data are likely to be or known to be non-uniformly distributed or in which there are many duplicate keys such that many keys will hash to the same value.

33) Let S and T be two arrays of n numbers that are already in nondecreasing order. Write an algorithm that finds the median of all 2n numbers whose time complexity is in Θ(lg n).

1. Find the medians of S_m of S and T_m of T.

2. If $S_m == T_m$, we found the overall median.

3. If $S_m > T_m$, the overall median is the median of the array composed of the left half of S and the right half of T. (recursive call)

4. If $S_m < T_m$, the overall median is the median of the array composed of the right half of S and the left half of T. (recursive call)

<u>Analysis</u>: The algorithm is similar to the binary search, in that it approximately halves the number of elements to search at each step (divide-and-conquer), and therefore its complexity is $\lg(2n) = \lg 2 + \lg n \in \Theta(\lg n)$.

34) Write a probabilistic algorithm that factors any integer using the functions prime and factor. Function prime is a boolean function that returns "true" if a given integer is a prime number and returns "false" if it is not. Function factor is a function that returns a nontrivial factor of a given composite integer. Analyze your algorithm, and show the results using order notation.

Finding a non-trivial factor of N can be done with a Monte Carlo algorithm called "Pollard's Rho":

```
int pollard_rho_factor(long N) {
        int a, b, s;
        a = random();
        b = a;
        a = f(a);
        b = f(f(b));
        while (a != b) {
                a = a % N;
                b = b % N;
                s = gcd(a – b, N);      //greatest common divisor
                if (s != 1 && s != N)
                        return s;       //this is a non-trivial factor!
                a = f(a);               //update a
                b = f(f(b));            //update b
        }
}

int f(x) {                              //other functions can be used
        return x*x+1;
}
```

Pollard, J. M. (1975), "A Monte Carlo method for factorization", BIT Numerical Mathematics 15 (3): 331–334

Primality testing can be performed with the Miller-Rabin algorithm:

```
bool MR_prime(long N, int nr_iter) {
        if ( N == 1 || (N!=2 && N%2==0) )     return false;
        if ( N == 2 || N == 3)                return true;
        long d = N - 1;
        while(d%2 == 0)         d/=2;
        for(int i=0; i<nr_iter; i++){
                long a  = rand() % (N-3) + 2;   //pick a random number between 2 and N-2
                long x  = pow(a, s) % N;
                if (x == 1)
                        continue;
                for (int i = 1; i <= s-1; i++){
                        x = x*x % N;
                        if (x == 1)
                                return false;
                        if (n == N-1)
                                continue;
                }
                return false;
          }
          return true;
}
```

Analysis:

- Pollard's algorithm has an average complexity $A(n) \in O(n^{1/4})$, and it has to be called at most lg n times.
- The Miller-Rabin algorithm decreases the probability that its argument be <u>not</u> prime by a factor of ¼ every time it iterates the for loop, so nr_iter is chosen such that $(¼)^{nr\_iter} <$ p-dependent-threshold; in most applications, nr_iter = 20 is sufficient. Using efficient implementations for the power and modulus operations, the M-R algorithm has complexity $O(nr\_iter \cdot \lg^3 N)$. It has to be called before each call to Pollard's algorithm, i.e. also < lg n times.
- The complexity of the entire algorithm is then $O(n^{1/4}\lg n + nr\_iter \cdot \lg^4 n) = O(n^{1/4}\lg n)$.

35) List the advantages and disadvantages of all the searching algorithms discussed in this chapter.

- Binary Search (BS):
  - Advantage: Very fast, and worst-case practically the same as average case (lg n).
  - Disadvantages: Data must be stored in array (which is not conducive to fast insertions and deletions), and array must be sorted.
- Interpolation Search (IS):
  - Advantage: Under the right conditions, it is even faster than BS!
  - Disadvantages:
    - As BS, data must be stored in array, and array must be sorted.
    - It has larger overhead than BS, due to more complex index calculations. It "goes logarithmic" or worse (worst-case is Q(n)) if distribution of keys is non-uniform. In experiments, it beats BS when the number of keys is very large (> 10^6) or the time to access the keys is large (external memory), and the distribution of the keys is close to uniform, but we are not aware of any useful real-life application that uses IS.
- B-Tree:
  - Advantages:
    - Appropriate for dynamic data, stored in a tree structure that permits fast insertions and deletions.
    - $\Theta(n)$ on average, and in worst-case.
    - Compared with a binary tree, the larger index of branching allows for "flatter" trees, which is good for external searches.
  - Disadvantages:
    - If data is not stored in a tree structure already, time must be spent building one.
    - Compared with a binary tree, the larger index of branching makes nodes bulkier, so it's slower if the tree is entirely stored in RAM.
- Hashing:
  - Advantages: Data need not be sorted! With a good hash function and few duplicates, it is the fastest search: Q(1)!.

- Disadvantages: Deals poorly with many duplicate keys. Does not allow searches on partial keys. No easy way to traverse all keys in sorted order. Needs to allocate memory for non-existing keys (although this can be alleviated with more advanced algorithms).

36) For each of the searching algorithms discussed in this chapter, give at least two examples of situations in which the algorithm is the most appropriate.

- Binary Search:
  - Small databases that store data mostly in RAM use a version of balanced BST called T-Tree (e.g. MySQL, Oracle Times-Ten).
  - Network routers use binary search to locate entries in their routing tables.
- Interpolation Search:
  - It's a direct competitor to binary search. It beats binary search only in applications where the number of keys is very large ($> 10^6$) or the time to access the keys is large (external memory), and the distribution of the keys is close to uniform. Dictionary lookup by a human is an example (accessing a key takes ≈ 1 second!).
  - Versions of interpolation search exist that can work efficiently (i.e. $\Theta(\log \log n)$ on average) with non-uniform distributions, if those distributions are known in advance.
- B-Tree:
  - Most modern relational database systems use a version of B-Trees (B+ Trees) for table indices (e.g. IBM DB2, Informix, PostgreSQL, MySQL, Microsoft SQL Server, Oracle, Sybase ASE, SQLite), along with other indexing methods. A B+ Tree index is more efficient than a hash index when <u>ranges</u> of records are to be retrieved, or when searches are performed on leftmost key characters rather than entire keys.
  - Many file systems (e.g. NTFS, ReiserFS, NSS, XFS, JFS, ReFS) use B-trees for metadata indexing.
- Hashing:
  - Most modern relational database systems use hashing for table indices, along with other indexing methods. A hash index is more efficient than a B-tree index when individual records are to be retrieved, and when the searches are performed on entire keys rather than partial keys.
  - The Linux Virtual File System (VFS) uses a hash table with overflow lists for its Information Nodes (inodes).