Algorithms and Advanced C Programming

Torino 2015

Y. Duan

Contents

Chapter 1

Complexity Analysis

1.1 Asymptotic Behavior

Let's start with a searching example where we want to find the maximum number in an array:

```
flag = 0;
for(i=0; i<N; i++){
    if(a[i] == x){
        flag = 1;
        break;
    }
}
// Check the value of flag
...
```

We assume each of the following operations takes the same time which is one unit time:

1. Assigning a value to a variable
2. Looking up the value of a particular element in an array
3. Comparing two values
4. Incrementing a value
5. Basic arithmetic operations such as addition and multiplication

The time used by this piece of code is:

```
flag = 0;                          // 1
for(i=0; i<N; i++){            // 1 + n + n, where n depends on how many
    times this loop executes
    if(a[i] == x){                  // n * 1
        flag = 1;                  // 1
        break;
    }
}
// Check the value of flag
...
```

For the worst case, the number we search for is at the end of the array. The total time used $f(n) = 3n + 3$. Here we only care about the ***asymptotic behavior*** of the worst case. That is, the asymptotic behavior of $f(n) = 3n + 3$ is described by the function $f(n) = n$.

1.1.1 Exercises

1. $f(n) = 5n + 12$
2. $f(n) = 109$
3. $f(n) = n^2 + 2n + 3$

1.2 Θ Notation

When we've figured out the exact such f asymptotically, we'll say that our program is $\Theta(g(n))$. For example, the above programs are $\Theta(1)$, $\Theta(n)$ and $\Theta(n^2)$. We call this function the time complexity or just complexity of our algorithm. Note that here we are talking about the worst case.

1.3 O Notation

$O(f(n)$ is an upper bound of $f(n)$. We have $\Theta(g(n)) \subseteq O(g(n))$. We can say $2n^2 + 3 = \Theta(n^2)$ and $2n^2 + 3 = O(n^2)$. While $2n^2 + 3 = O(n^3)$ is also correct even if it is not quite meaningful in the complexity analysis. While $2n^2 + 3 = \Theta(n^3)$ is wrong.

1.3.1 Exercises

1. A $\Theta(n)$ algorithm is $O(n)$
2. A $\Theta(n)$ algorithm is $O(n^2)$
3. A $\Theta(n)$ algorithm is $O(1)$
4. A $\Theta(1)$ algorithm is $O(1)$
5. A $O(n)$ algorithm is $\Theta(1)$

1.4 An Example of Insertion Sort

```
for(i=1; i<n; i++) {
     x = A[i];
     j = i - 1;
     while (j >= 0 && x < A[j]){
          A[j+1] = A[j];
          j--;
     }
     A[j+1] = x;
}
```

$1 + n + n$
$n - 1$
$n - 1$
$2\sum_{k=2}^{n}$
$\sum_{k=1}^{n-1}$
$\sum_{k=1}^{n-1}$

$n - 1$

By summing up we have

$$T(n) = \frac{3}{2}n^2 + \frac{9}{2}n - 3$$

We can say that $T(n) = \Theta(n^2)$, $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$.

Chapter 2

Union Find

Given several objects, we want to connect them together. For example, a network consists of 10 nodes as shown in Figure 2.1. Each node has been given a number. We may add connections represented by pairs of numbers in the network. When adding such connections, we want to determine whether we need to establish a new direct connection for p and q to be able to communicate or whether we can use existing connections to set up a communications path. The operation of connecting two nodes is called **union** and the one of testing if two nodes are already connected is called **find**.

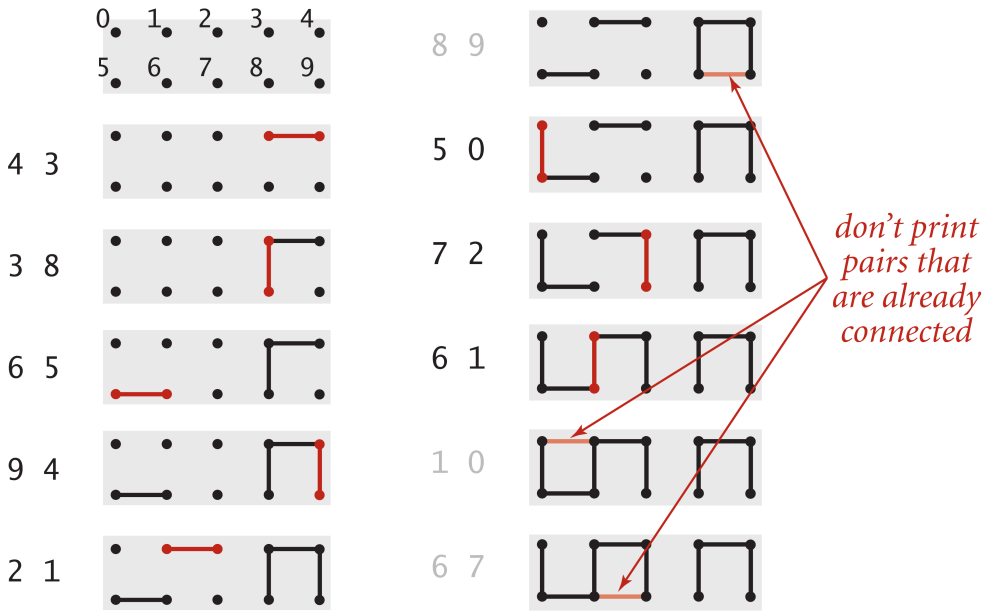
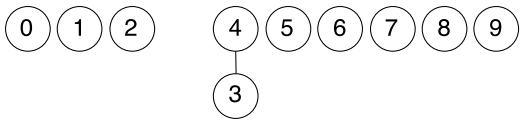


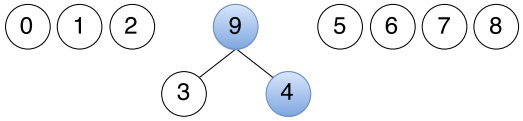
Figure 2.1: An example of dynamic connectivity in networks.

2.1 Quick Find

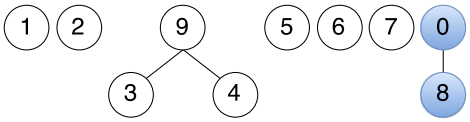
3-4 0 1 2 4 4 5 6 7 8 9



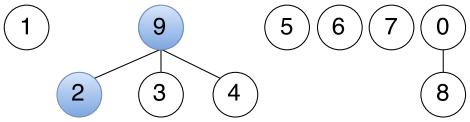
4-9 0 1 2 9 9 5 6 7 8 9



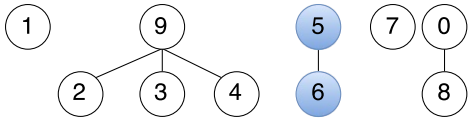
8-0 0 1 2 9 9 5 6 7 0 9



2-3 0 1 9 9 9 5 6 7 0 9

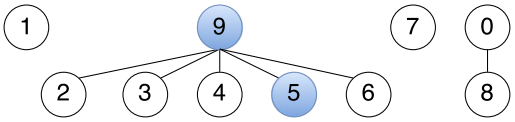


5-6 0 1 9 9 9 6 6 7 0 9

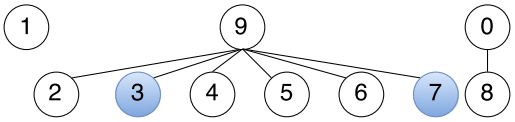


2-9 Do nothing 2 and 9 are already connected

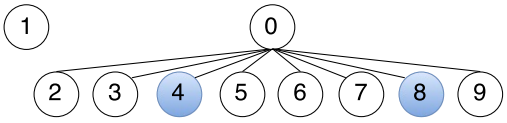
5-9 0 1 9 9 9 9 9 7 0 9



7-3 0 1 9 9 9 9 9 9 0 9

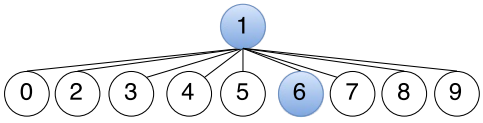


4-8 0 1 0 0 0 0 0 0 0 0



0-2 Do nothing 0 and 2 are already connected

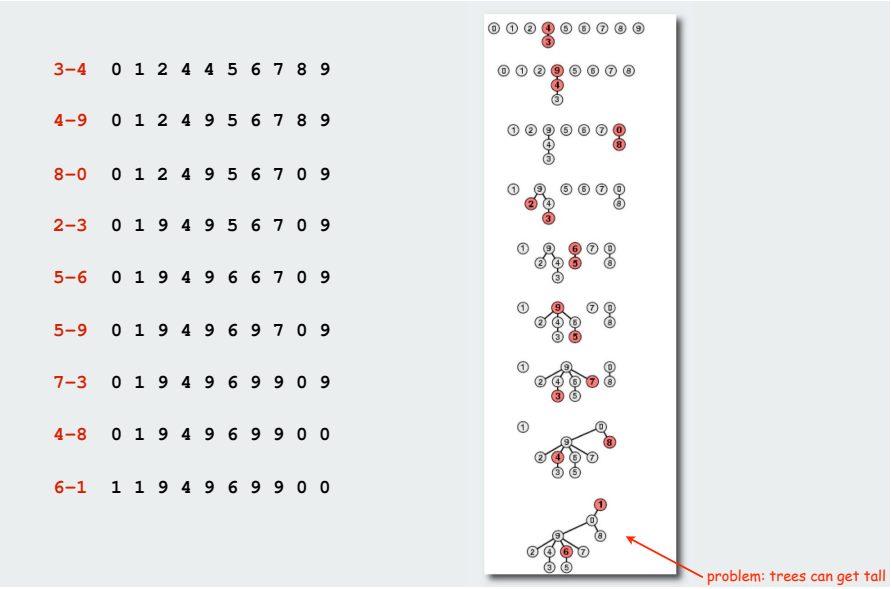
6-1 1 1 1 1 1 1 1 1 1 1



5-8 Do nothing 5 and 8 are already connected

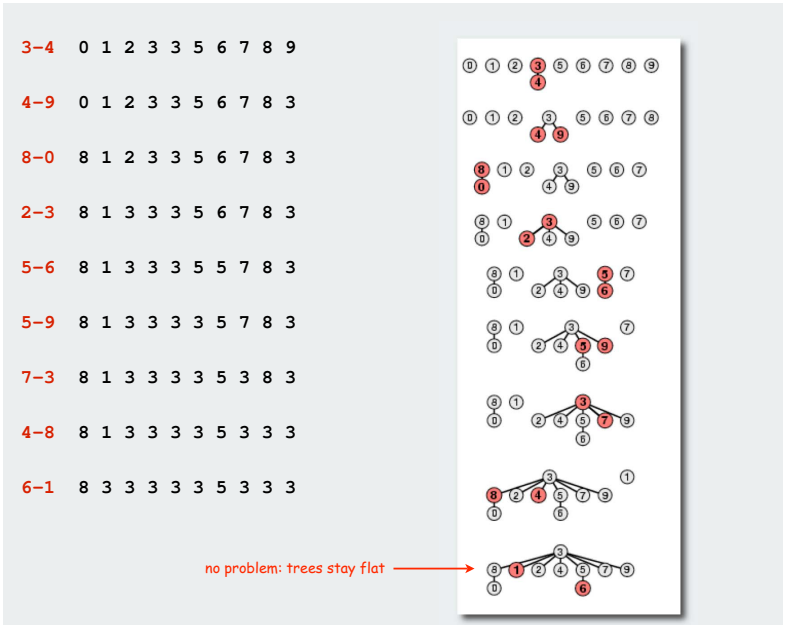
2.2 Quick Union

Similarly we have



2.3 Weighted Quick Union

In each step, we add the smaller tree to the bigger one.



2.4 Complexity Analysis

Algorithm	Union	Find
Quick Find	N	1
Quick Union	tree height	tree height
Weighted Quick Union	lgN	lgN

Chapter 3

Iterative Sorting Algorithms

3.1 Bubble Sort

The idea is to move the biggest value to one side of the array.

```
for(i=0; i<n-1; i++) {
    for (j = 0; j < n-1-i; j++)
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
    }
}
```

Optimized Bubble Sort

During the execution, if it finds that the left sub-array is already sorted, the sorting can stop and the whole array has been sorted.

```
flag = 1;
for(i=0; i<n-1 && flag==1; i++) {
    flag = 0;
    for (j = 0; j < n-1-i; j++)
    if (A[j] > A[j+1]) {
        flag = 1;
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
    }
}
return;
```

Here is an example of bubble sort:

68	21	10	67	8	41	70	47	48	61
21	10	67	8	41	68	47	48	61	70
10	21	8	41	67	47	48	61	68	70
10	8	21	41	47	48	61	67	68	70
8	10	21	41	47	48	61	67	68	70
8	10	21	41	47	48	61	67	68	70
8	10	21	41	47	48	61	67	68	70
8	10	21	41	47	48	61	67	68	70
8	10	21	41	47	48	61	67	68	70
8	10	21	41	47	48	61	67	68	70

Complexity of bubble sort:

$$T(n) = (n-1) + (n-2) + (n-3) + ... + 2 + 1 = O(n^2)$$

Bubble sort is in-place, stable.

3.2 Selection Sort

The idea of selection sort is to always exchange the current leftmost value with the smallest value among the unsorted ones.

```
for(i=0; i<len; i++){
        p = i;
        for(j=i+1; j<len; j++){
                if(a[p] > a[j])
                        p = j;
        }
        temp = a[p];
        a[p] = a[i];
        a[i] = temp;
}
```

Here is an example of selection sort:

82	**3**	76	25	62	43	96	85	42	83
3	**82**	76	**25**	62	43	96	85	42	83
3	25	**76**	82	62	43	96	85	**42**	83
3	25	42	**82**	62	**43**	96	85	76	83
3	25	42	43	**62**	82	96	85	76	83
3	25	42	43	62	**82**	96	85	**76**	83
3	25	42	43	62	76	**96**	85	**82**	83
3	25	42	43	62	76	**82**	85	96	83
3	25	42	43	62	76	82	**83**	96	85
3	25	42	43	62	76	82	83	**85**	96
3	25	42	43	62	76	82	83	85	96

Complexity of selection sort:

$$T(n) = (n-1) + (n-2) + (n-3) + ... + 2 + 1 = O(n^2)$$

Selection sort is in-place, unstable.

3.3 Counting Sort

```
void CountingSort(int A[], int n) {
    int i, j, *B, C[k];
    B = malloc(n*sizeof(int));
    for (i=0; i<k; i++)
        C[i] = 0;
    for (i=0; i<n; i++)
        C[A[i]]++;
    for (i=1; i<k; i++)
        C[i] += C[i-1];
    for (i=n-1; i>=0; i--) {
        B[C[A[i]]-1] = A[i];
        C[A[i]]--;
    }
    for (i=0; i<n; i++)
        A[i] = B[i];
}
```

Define 4 arrays, A, B, C: A is the original array that is to be sorted; B is the final sorted array; C is an auxiliary array that will be explained later.

A

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

Assume all the numbers in A are non-negative and the maximum value is k. We declare an array C that has $(k+1)$ elements, where all the elements are 0 initially, as shown in C_1. Then we let the each element of C equal to the number of occurrences of the corresponding numbers, as shown in C_2. Given the occurrences, we let each element in C be the accumulation of all the numbers before.

C_1

0	1	2	3	4	5
0	0	0	0	0	0

C_2

0	1	2	3	4	5
2	0	2	3	0	1

C_3

0	1	2	3	4	5
2	2	4	7	7	8

We can notice that the values in C2 indicate the last positions of the corresponding numbers in the sorted array. We started with positioning the last number of array A in array B. As the last number of A is 3, we search 3 in C2 and we got C2[3] = 7. So we put 3 in the B[C2[3]-1] which is B[6] and then decrease C2[3] by 1. Note that A[5] is also equal to 3. As we have already decreased C2[3], this time 3 should be positioned in B[C2[3]-1], i.e., B[5]. Once we have finished placing the numbers into B, we can copy all the numbers of B into A.

Final A

0	1	2	3	4	5	6	7
0	0	2	2	3	3	3	5

It is easy to obtain that the complexity of Counting sort is $O(n)$.
Counting sort is not in-place, while stable.

3.4 Shell sort

It sorts the numbers between two of which the distance is h firstly. Then it decreases h and sort the numbers again until h reaches to 1. Let's set "$h_{i-1} = 3 * h_i + 1, i \in \{0, 1, 2, 3\}$" as an

example. The sequence of h is $1, 4, 13, 40 \cdots$. Note that each h should be less than the length of the array. In our example, there are 20 numbers in the array, so we set h_1 to $3 * 4 + 1 = 13$.

Initial array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
47	48	61	30	27	88	81	90	7	28	1	50	87	68	21	10	67	8	41	70

We divide this array into 2 parts, whose first numbers are 13 elements far from each other.

47 68
48 21
61 10
30 67
27 8
88 41
81 70
90
7
28
1
50
87

Then we sort the numbers in each line and we got the following array.

47 21 10 30 8 41 70 90 7 28 1 50 87 68 48 61 67 27 88 81

As next h in the sequence is 4, we divide the numbers into 5 parts, keep the distance of the first numbers in each part equal to 4.

47 8 7 87 67
21 41 28 68 27
10 70 1 48 88
30 90 50 61 81

As before, we sort the numbers in each line and we obtain:

7 21 1 30 8 27 10 50 47 28 48 61 67 41 70 81 87 68 88 90

Finally h reaches to 1 and we will sort the entire array.

1 7 8 10 21 27 28 30 41 47 48 50 61 67 68 70 81 87 88 90

Properly choosing the sequence of h can reduce the complexity of the Shell sort algorithm. The following tables shows some typical sequences along with their corresponding complexity.

$h_{i-1} = 3 * h_i + 1$	$O(n^{\frac{3}{2}})$
$h_1 = 1, h_i = 4^(i + 1) + 3 * 2^i + 1$	$O(n^{\frac{4}{3}})$
$h_i = 2^i, i = 0, 1, 2 \cdots$	$O(n^2)$

Shell sort is in-place but unstable.

3.5 Exercise

Sort the following numbers using the **bubble sort**, **selection sort** and **shell sort**:

61, 109, 149, 111, 34, 2, 24, 119, 122, 125, 27, 145

Chapter 4

Recursion and Sorting

4.1 An Starting Example

✿ *Do it now*
Consider a function that prints an integer number n.

```
void print(int n)
{
    printf("%d\n", n);
}
```

Have a look at the following code. This code will print the numbers $n, n-1, n-2, \ldots, 2, 1$.
❀**Think**: *What if we want to print the sequence:* $n, n-2, n-4, ..., 1$ *or* 0*?*

```
void rprint(int n)
{
    if(n == 0)
        return;
    printf("%d\n", n);
    rprint(n-1);
}
```

The following piece of codes calculates the sum of the integer numbers from 1 to 100.

```
int sum(int n)
{
    if(n == 1)
        return 1;
    else
        return n + sum(n-1);
}
```

Note that *else is not necessary in the codes above.* ❀**Think**: *How to use recursion to calculate the factorial?*

```
int sum(int n)
{
    if(n == 1)
        return 1;
    return n + sum(n-1);
}
```

4.2 Complexity in Recursion

Search a number in a sorted array recursively.

```
int binarySearch(int *a, int l, int r, int key){
    int mid;
    if (r < l)
        return 0;

    mid = (l + r) / 2;
    if (a[mid] == key)
        return 1;
    else if (key < a[mid])
        return binarySearch(a, l, mid-1, key);
    else
        return binarySearch(a, mid+1, r, key);
}
```

This binary search uses the **Divide**, **Conquer** and **Combine** concept.

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Complexity:

Divide Dividing the array into 2 sub-arrays: $\Theta(1)$

Conquer $T(\frac{n}{2})$

Combine No cost in combining.

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

which can be rewritten as:

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

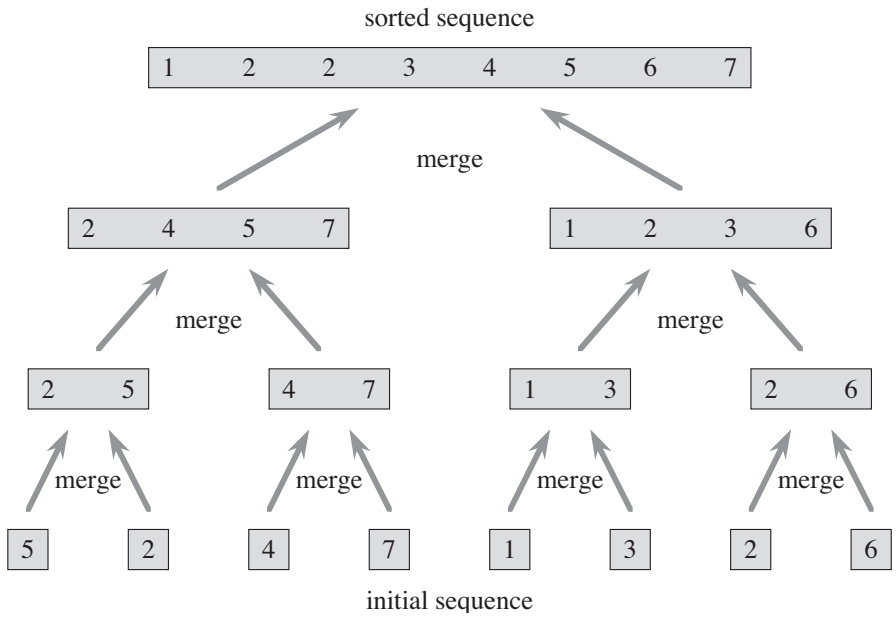
To obtain $T(n)$, unfold it step by step and find the regular pattern:

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + 1 \\ &= T\left(\frac{n}{4}\right) + 1 + 1 \\ &= T\left(\frac{n}{8}\right) + 1 + 1 + 1 \\ &= T\left(\frac{n}{16}\right) + 1 + 1 + 1 + 1 \\ &= T(1) + \underbrace{1 + 1 + \ldots + 1}_{\log_2 n} \end{aligned}$$

Therefore,

$$T(n) = \log_2 n + 1 = \Theta(\lg n)$$

4.3 Merge Sort



Merge sort uses the **Divide**, **Conquer** and **Combine** concept.

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

The codes of Merge sort:

```
void MergeSort(int A[], int l, int r)
{
    int q;
    if (r <= l)
        return;
    q = (l + r)/2;
    MergeSort(A, l, q);
    MergeSort(A, q+1, r);
    Merge(A, l, q, r);
}
```

```
void Merge(int A[], int l, int q, int r)
{
    int i, j, k;
    int B[ARRAY_LENGTH];
    i = l;
    j = q+1;
    for (k = l; k <= r; k++)
        if (i > q)
            B[k] = A[j++];
```

```
        else if (j > r)
            B[k] = A[i++];
        else if (A[i] <= A[j])
            B[k] = A[i++];
        else
            B[k] = A[j++];
    for (k = l; k <= r; k++)
        A[k] = B[k];
    return;
}
```

Complexity Analysis

Divide The divide step just computes the middle of the subarray, which takes constant time: $\Theta(1)$.

Conquer : We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.

Combine : We have already noted that the *Merge()* procedure on an n-element subarray takes time $\Theta(n)$.

Thus we have the complexity for the array size n:

$$T(n) = \Theta(1) + 2T\left(\frac{n}{2}\right) + \Theta(n)$$

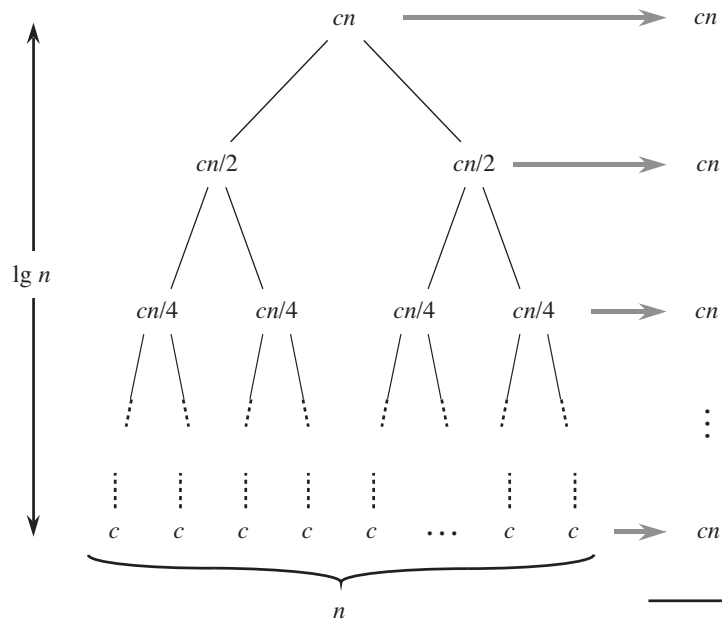
which can be rewrite as

$$T(n) = a + 2T\left(\frac{n}{2}\right) + cn$$

and further

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

We derive the following *recursion tree*. The fully expanded tree has $\log n + 1$ levels (the height is $\log n$):



Total: $cn \lg n + cn$

Therefore,

$$T(n) = \Theta(n \log n)$$

Merge sort is not in-place but stable.
❀**Think**: *Use substitution (unfolding) method to evaluate the complexity.*

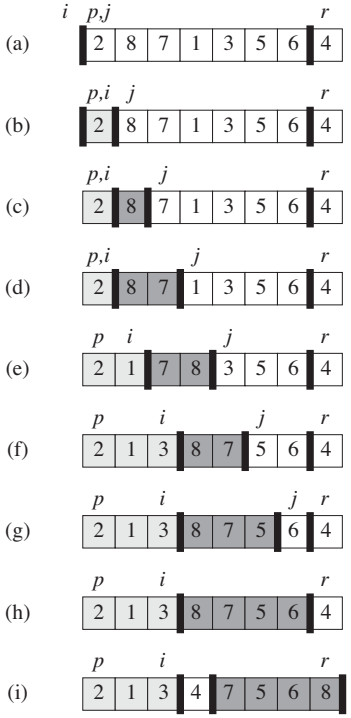
4.4 Quick Sort

```
void quicksort(int * const a, int l, int r)
{
    int pivot, i;
    if(l >= r)
        return;

    /* 1 partition */
    pivot = a[r];
    i = l;
    for (int j = l; j < r; ++j) {
        if(a[j] < pivot){
            swap(a+i, a+j);
            i++;
        }
    }
    swap(a+i, a+r);

    /* 2 quick sort on left */
    quicksort(a, l, i-1);
    /* 3 quick sort on right */
    quicksort(a, i+1, r);
}
```

An example of *partition*:



A full example:

26	75	12	93	46	35	92	33	66	95	72	73	86	55	52	13
12	13	26	93	46	35	92	33	66	95	72	73	86	55	52	75
12	13	26	46	35	33	66	72	73	55	52	75	86	95	93	92
12	13	26	46	35	33	52	72	73	55	66	75	86	95	93	92
12	13	26	33	35	46	52	72	73	55	66	75	86	95	93	92
12	13	26	33	35	46	52	72	73	55	66	75	86	95	93	92
12	13	26	33	35	46	52	55	66	72	73	75	86	95	93	92
12	13	26	33	35	46	52	55	66	72	73	75	86	95	93	92
12	13	26	33	35	46	52	55	66	72	73	75	86	92	93	95
12	13	26	33	35	46	52	55	66	72	73	75	86	92	93	95
12	13	26	33	35	46	52	55	66	72	73	75	86	92	93	95

❀**Think**: *What if we want to select another element as the pivot?*

Complexity Analysis

In worst case, during the partitioning, the array is always divided into two arrays where one of them has the length 1. In such a case, we have

$$T(n) = T(n-1) + T(1) + \Theta(n)$$

or

$$T(n) = T(n-1) + \Theta(n)$$

from which we can know that

$$T(n) = \Theta(n^2)$$

In best case, during the partitioning, the array is always cut in half. In this case, we have

$$T(n) = 2T(n/2) + \Theta(n)$$

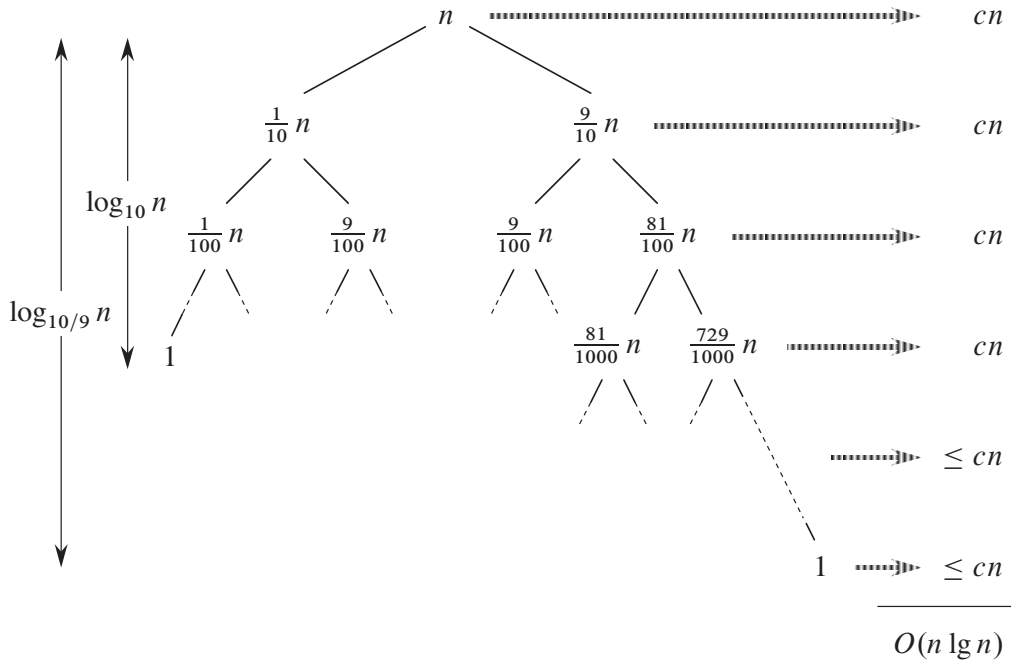
which gives a complexity of

$$T(n) = \Theta(n \lg n)$$

The average-case running time of quick sort is much closer to the best case than to the worst case

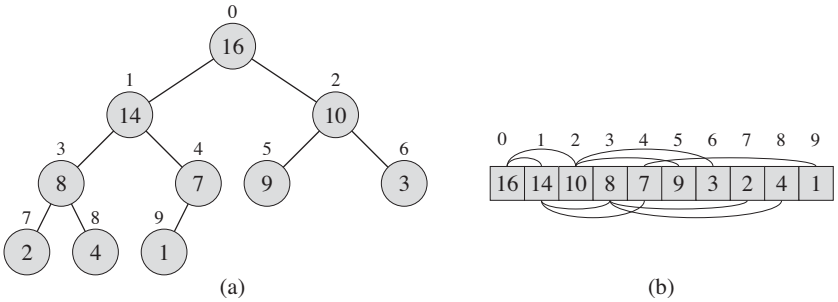
$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + cn$$

If we draw the recursion tree



4.5 Heap

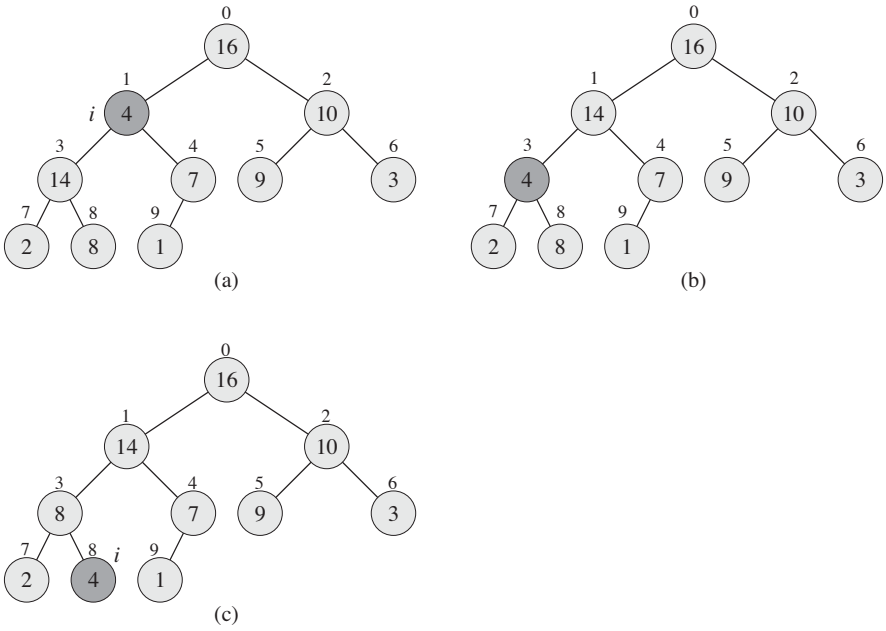
A heap is an array that can be represented in a nearly complete binary tree. A max heap is a heap where each node is larger than both of children.



For the heap there are two main operations: **heapify** and **build heap**.

Heapify

Heapify adjusts a local unordered node and arrange it to the right position. Here is an example where we execute `heapify(heap, 1)`.



The code of heapify operation is showed in below. Note that for a node i, if it has a parent, the index of its parent is $\lfloor (i-1)/2 \rfloor$. The left child and right child of node i are $2i+1$ and $2i+2$ respectively. To make the code more clear, we define `PARENT`, `LEFT` and `RIGHT` for the parent, left child and right child of a node.

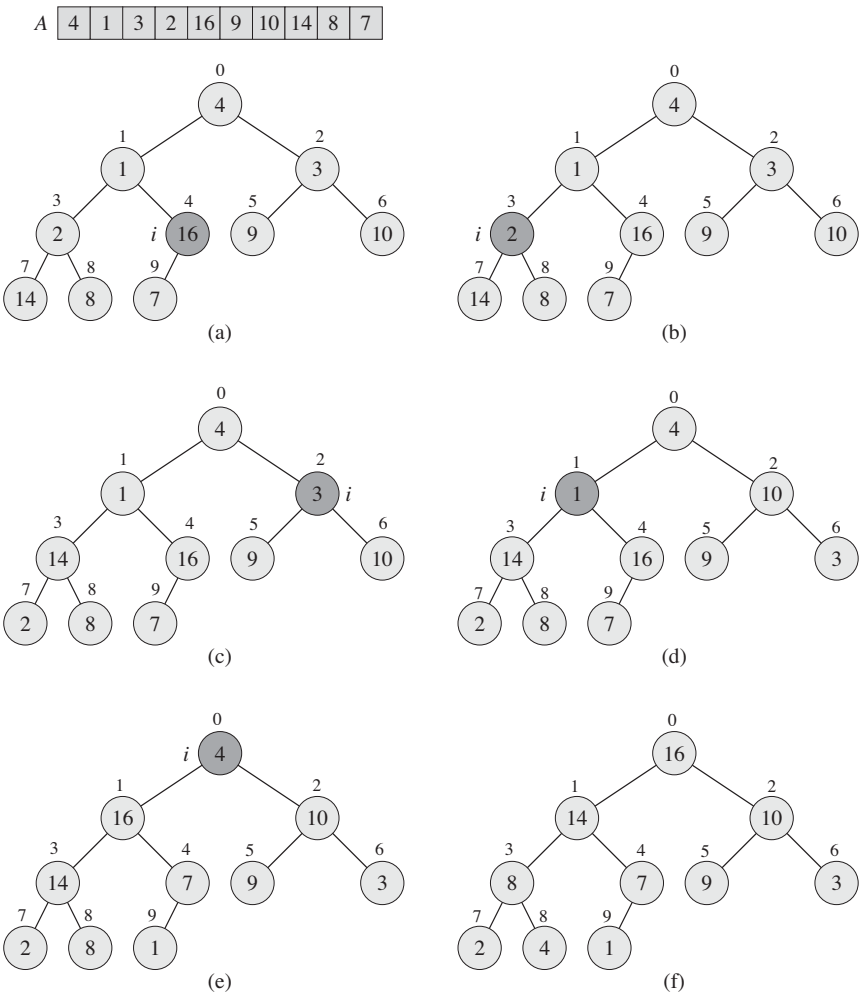
```
#define PARENT(i) (i-1)/2
#define LEFT(i) 2*i+1
#define RIGHT(i) 2*i+2
```

```
void heapify(int * const a, int i){
    int l, r, largest;
    l = LEFT(i);
```

```
    r = RIGHT(i);
    largest = i;
    if(l<heapsize && a[l]>a[largest])
        largest = l;
    if(r<heapsize && a[r]>a[largest])
        largest = r;
    if(largest != i){
        exchange(a, i, largest);
        heapify(a, largest);
    }
}
```

Build heap

Build heap constructs a heap by ordering the elements in an array. The following example turned the original array A into a max heap.

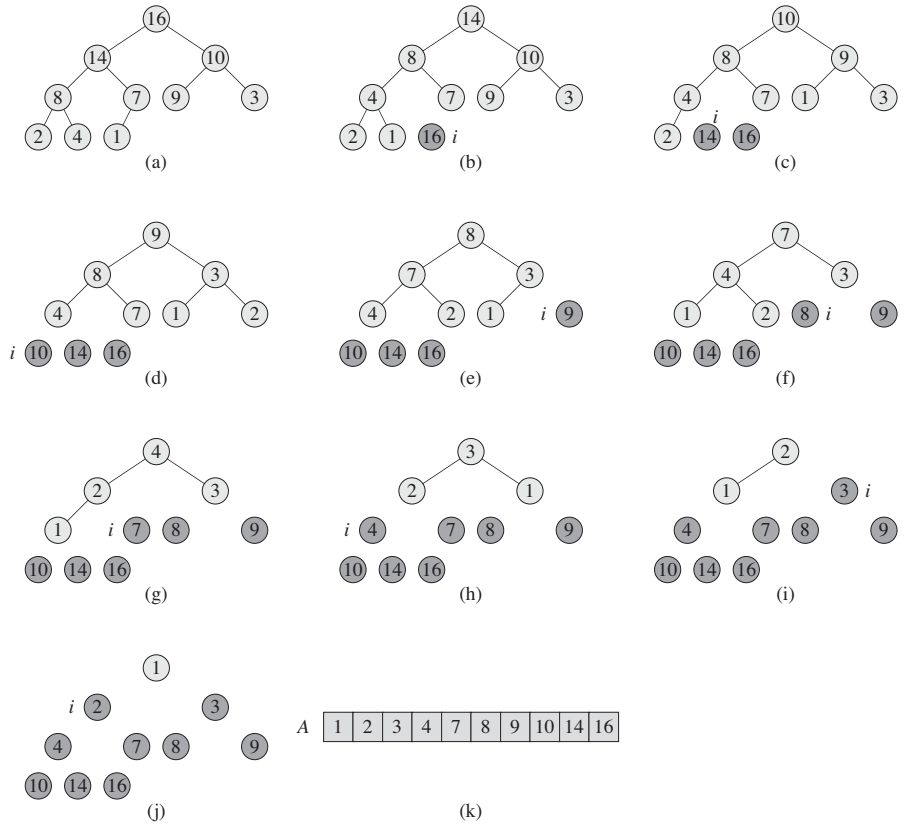


buildHeap does heapify on each node has no children starting from the last one (down) to the first one(top). The index of the last one among the no-children nodes is $n/2-1$ for a heap that has n nodes. The code is below.

```
void buildHeap(int * const a){
    int i;
    for(i=N/2-1; i>=0; i--)
        heapify(a, i);
}
```

4.6 Heap Sort

Heap sort works based on heap. It first turns an unsorted array into a heap. Then it exchanges the top node with the last node at the bottom and decrease the heap size by 1. As the new top node violets the definition of a heap, we perform *heapify* on the top node and the array (except the last one) is a heap again. We go on with this procedure until we reach the top node, after which, the array is sorted.




```
void HeapSort(int * const a){
    int i;
    buildHeap(a);
    for(i=N-1; i>=0; i--){
        exchange(a, i, 0);
```

```
        heapsize--;
        heapify(a, 0);
    }
    heapsize = N; // N is original heap size
}
```

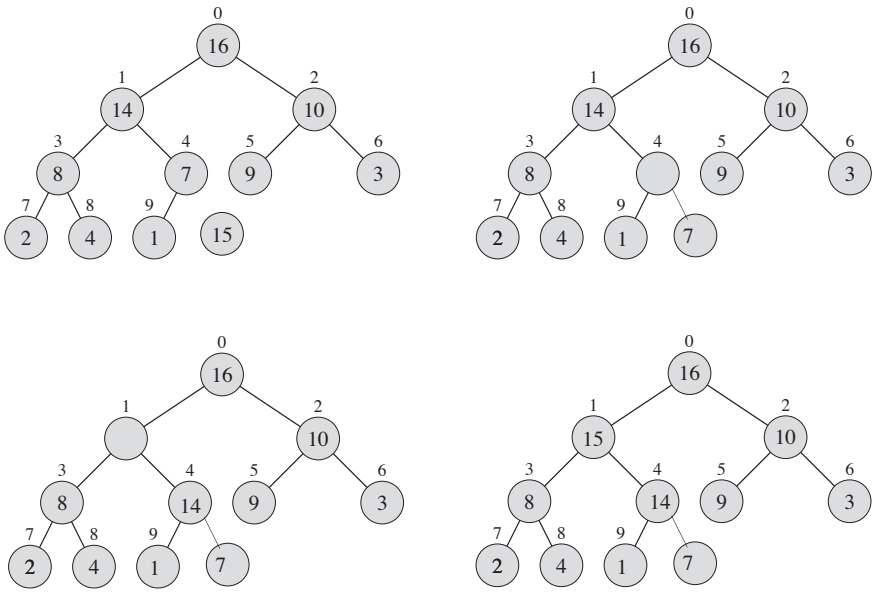
4.7 Priority Queue

A *priority queue* is a data structure for maintaining a set S of elements, each with an associated value called a priority. We have four operations for a priority queue:

1. `PQInsert(PQ, Item)`
2. `PQShowMax(PQ)`
3. `PQExtractMax(PQ)`
4. `PQChange(PQ, pos, Item)`

4.7.1 PQInsert

`PQInsert(PQ, Item)` inserts a new item `item` to the priority queue `PQ`. It first adds the item to the last position of the queue (as a new leaf). Then we keep comparing it with its ancestors till the right position. If it is a greater than an ancestor, we "pull down" the ancestor. The following example inserts an item with priority equal to 15.



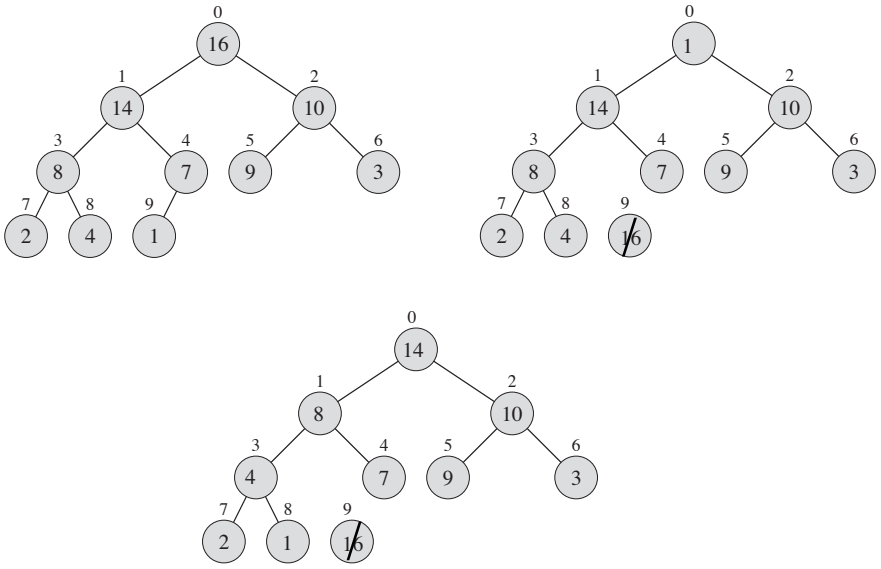
❀**Think**: *What is complexity of this operation?*

4.7.2 PQShowMax

The item with the highest priority which is exactly `pq[0]`.

4.7.3 PQExtractMax

Recall the heap sort procedure, given a heap, we swap the top node with the last node at the bottom to "extract" it. Here have the same method: swap and heapify on the new top node to maintain a heap.

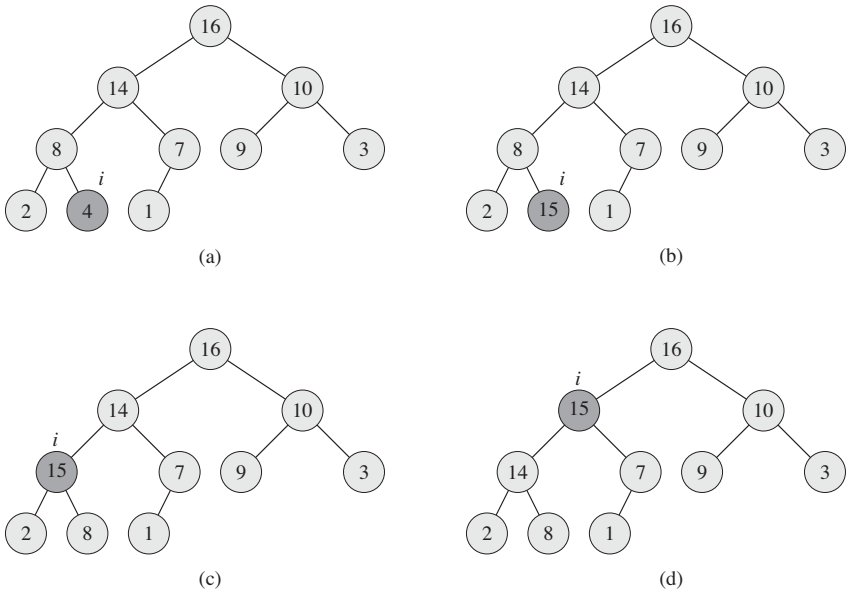


❀**Think**: *What is complexity of this operation?*

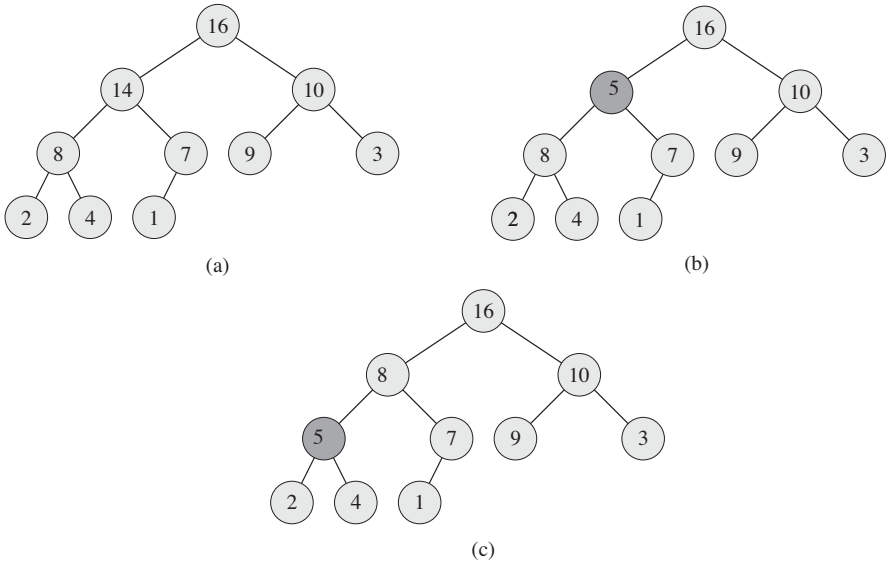
4.7.4 PQChange

`PQChange(PQ, pos, Item)` changes the item at postition `pos` to the new priority `Item`. First it should check if the new priority is larger than the one of its parent. If yes, move it up. Then it will perform a heapify procedure on this node (provided that the heaify procedure has already checked at the beginning if its value is larger that the children).

In the following example, we change the item at position 8 to new value 15, which is larger than the one of its parent.



Another example below shows the procedure to change the item at position 1 to new value 5.



Chapter 5

Greedy Algorithm

5.1 Huffman Code

Suppose a text file contains 100 characters from a to f. Different characters appear for different times, as shown in Table 5.1. To represent the file, we can design a binary character code where each character is represented by a unique binary string. The code can have fixed lengths or variable lengths.

Table 5.1: Fixed-length and variable-length binary code for a given text with characters from a to z. Different characters have different occurrences.

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length	000	001	010	011	100	101
Variable-length	0	101	100	111	1101	1100

If we use the fixed-length binary code encoding, the size of the file would be $100 \times 3 = 300$bits. Instead, if we use the variable-length one, the size would be

$$45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4 = 224\text{bits}$$

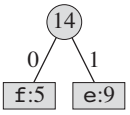
Apparently, the second way of representing is compressing the text with respect to the first one. In fact, this is the optimal character code for this file, which we shall see later.

5.1.1 Generating Huffman Code

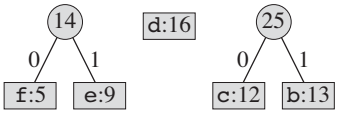
Always combine the lowest two numbers together.

(a) f:5 e:9 c:12 b:13 d:16 a:45

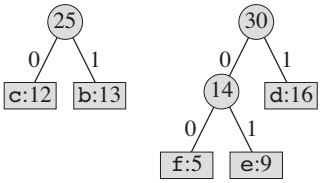
(b) c:12 b:13 14 d:16 a:45



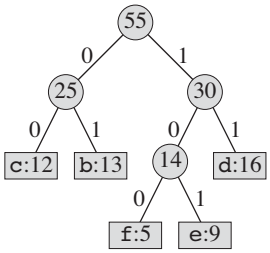
(c) 14 d:16 25 a:45



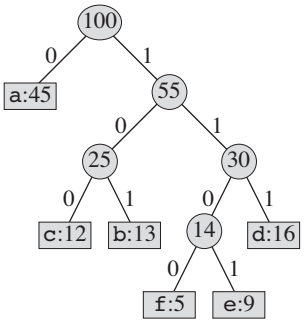
(d) 25 30 a:45



(e) a:45



(f)



Chapter 6

Data Structures

6.1 Hash Table

Assume we want to keep track of the number of daily flights at the airports in Europe . Each airport is identified by a 3-letters code, e.g., `LHR`, `CDG`, `MUC`etc. We may want to list a table to store the information, as shown in Table 6.1. When we insert the airport, we should keep in mind that it should also be easy to retrieve the data of a airport. While the total number of possible permutations of three letters are 26^3, 500 of them are already enough to represent all the airports in Europe. Thus the table can be composed of 500 lines and the airports should be mapped to the 500 lines. Here we denote by $h(k)$ the code of each airport. We design a function $h(k)$ that takes the airport code and map it to the line number. This function can be called the **hash function** and the table can be called the hash table. The airport code here is called **key**. Since the airport codes are *randomly* distributed, it is inevitable that many airports would be mapped to the same line number, given a large universal set and a small final set. This phenomenon is called **clustering**. A good hash function should distribute the hash values and reduce clustering. In the following we shall see three types of functions.

Table 6.1: Airports and number of flights.

LHR	5000
CDG	3000
MUC	4000

6.1.1 Linear probing

Assume the size of the hash table(i.e., number of lines) is m. We can first translate the key to a integer number. There are many ways transform a string to a number. One way to achieve this is just to take the string as a radix-26 representation of a number. For example, `LHR` can be translated to $11 * 26^2 + 7 * 26 + 17 * 1$. To map a key k to a line number, the simplest way is to take the remainder of k divided by m, which implies, $h(k) = k \bmod m$. For instance, if the hash table size m is 14, $h(15) = 1$ and $h(200) = 4$. However, we may notice that $h(312) = 4$. We denote by i the number of turns that we want to insert the keys that would be mapped to the same hash value. The final hash function is

$$h(k, i) = (k + i) \bmod m$$

.

In the following we give an example to show how it works. Suppose we want to insert the keys $k_1 = 102, k_2 = 202, k_3 = 302$ into a table whose size is 20. Firstly we have $h(k_1, 0) = 2$. When inserting k_2, we have $h(k_2, 0) = 2$, which is apparently not allowed. We further compute $h(k_2, 1) = 3$. Thus k_2 should be inserted into the line 3. To insert k_3, we have $h(k_3, 0) = 2$. Similarly, we have to compute $h(k_3, 1) = 3$ till $h(k_3, 2) = 4$.

The Linear probing solution suffers from a *primary clustering* problem.

6.1.2 Quadratic probing

The Quadratic probing hash function is

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where $h'(k)$ is called the *auxiliary hash function* . This solution results in a milder clustering, which is called *secondary clustering*.

6.1.3 Double hashing

Double hashing uses the following form

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

. For example, if we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. We can insert the keys (79, 69, 98, 72, 14, 50) into the positions in the following way.

0	1	2	3	4	5	6	7	8	9	10	11	12
	79			69	98		72		14		50	

The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire keys. If m is power of 2, we design a h_2 such that it always returns an odd value. Another solution is to let m be prime and to design h_2 in order that it is always smaller than m. For example, a pair of h_1 and h_2 can be

$$h_1(k) = k \bmod m$$
$$h_2(k) = 1 + (k \bmod m')$$

, where m' is slightly smaller than m (e.g., $m - 1$).

In the following we show an example where it is required to design a double hashing function.

> Given the sequence of keys I B U X E G Z H Y, where each character is identified by its index in the English alphabet (A=1, ..., Z=26), draw the final configuration of an initially empty hash table of size 19 where insertion of the previous sequence occurs character by character. Assume open addressing with double hashing. Define suitable hash functions. Show relevant intermediate steps.

Let

$$h_1(k) = k \bmod 19$$
$$h_2(k) = 1 + (k \bmod 18)$$

Then we start insert the keys into the table. We obtained the following result:

Keys	I	B	U	X	E	G	Z	H	Y
Key value	9	2	21	24	5	7	26	8	25
Hash Value	9	2	6	5	11	7	16	8	14

6.2 List

In this section we should learn in the implementation of list in C.

1. How to implement a list in C
2. How to generate a list in C
3. The operations of list:
   (a) visit all the nodes in the list
   (b) add(append) a value to a list
   (c) search for a value
   (d) insert a node
   (e) change the value of a node
   (f) remove a node

Chapter 7

Graph

- Vertex (Vertices)
- Edge

7.1 Depth First Searching

7.1.1 DFS

7.1.2 Tree labling

7.2 Breadth First Searching

7.3 Bridge

7.4 Articulation

7.5 Topological Sort

7.5.1 Reverse topological sort

7.6 Strongly Connected Component

7.7 Minimum Spanning Tree

7.7.1 Kruscal's algorithm

7.7.2 Prim's algorithm

7.8 Single Source Shortest Path

7.8.1 Dijkstra's algorithm

Negative loops

7.8.2 Bellman-Ford's algorithm

Chapter 8

Tree

8.1 Binary Search Tree

8.1.1 Binary tree implementation

8.1.2 Visiting binary tree

1. Insert a new node

8.1.3 Visiting binary tree

Pre-order V-L-R

In-order L-V-R

Post-order L-R-V

8.2 Binary Expression Tree

8.3 General Trees

8.3.1 Implementation

8.3.2 Visiting trees