2 Exercises and Solutions

Most of the exercises below have solutions but you should try first to solve them. Each subsection with solutions is after the corresponding subsection with exercises.

2.1 Sorting and searching: exercises

1. You must find 1000 most expensive items from an unsorted price list containing 10^7 different items. Two schemes of solution are as follows.

   - Scheme **A**: repeat 1000 times the sequential search (with the linear complexity $O(n)$).
   - Scheme **B**: convert the list into an array (the same complexity $O(n)$ as for search), then sort the array (complexity $O(n \log n)$) and fetch 1000 top items.

   Which scheme would you prefer assuming that searching for 100 items in the unsorted list takes 0.1 millisecond (ms) and sorting of 100 items also takes 0.1 ms? Time for fetching data should not be taken into account.

2. Processing time of InsertionSort is $c \cdot n^2$. To merge k pre-sorted subarrays that contain together n items you have to compare the k top items in all the subarrays to choose the current maximum item and place it into the sorted array (assuming that all the items have to be sorted in ascending order). Therefore, the time for merging is proportional to $(k-1) \cdot n$. Let the processing time of the merge be $c \cdot (k-1) \cdot n$ where the scale factor has the same value as for InsertionSort. Analyse whether it is possible to accelerate sorting of n items in the following way:

   - Split the initial array of size n into k subarrays of size n/k. The last subarray may be longer to preserve the total number of items n but do not go in such details in your analysis.
   - Sort each subarray separately by InsertionSort.
   - Merge the sorted subarrays into a final sorted array.

   If you have found that the acceleration is possible, then find the optimum value of k and compare the resulting time complexity of this sorting algorithm to that of InsertionSort and MergeSort.

3. The processing time of a sorting algorithm is described by the following recurrence equation (c is a positive constant):

   $$\begin{array}{rcl} T(n) & = & 3T(n/3) + 2cn; \\ T(1) & = & 0 \end{array}$$

   Solve this equation to derive an explicit formula for $T(n)$ assuming $n = 3^m$ and specify the Big-Oh complexity of the algorithm.

4. Find out whether the above algorithm is faster or slower than usual Mergesort that splits recursively an array into only two subarrays and then merges the sorted subarrays in linear time cn.

5. Time complexity of sorting n data items with `insertionSort` is $O(n^2)$, that is, the processing time of `insertionSort` is $c_\text{s} \cdot n^2$ where c_s is a constant scale factor.

   Time complexity of merging k subarrays pre-sorted in ascending order and containing n items in total is $O(k \cdot n)$. The processing time is $c_\text{m}(k-1) \cdot n$ because at each step $t = 1, 2, \ldots, n$ you compare k current top items in all the subarrays (that is, $(k-1)$ comparisons), pick up the maximum item, and place it in position $(n-t+1)$ in the final sorted array.

   You are going to save time by splitting the initial array of size n into k smaller subarrays, sorting each subarray by `insertionSort`, and merging the sorted subarrays in a single sorted array.

   Let the scale factors c_s and c_m be equal: $c_\text{s} = c_\text{m} = c$. Analyse whether you can actually accelerate sorting of n items in the following way:

   - Split the initial array of size n into k subarrays of size $\frac{n}{k}$. The last subarray may be longer to preserve the total number of items n but do not go into such detail in your analysis.
   - Sort each subarray separately by InsertionSort.
   - Merge the sorted subarrays into a final sorted array.

   If the acceleration is possible, then find the optimum value of k giving the best acceleration and compare the resulting time complexity of the resulting sorting algorithm to that of InsertionSort and MergeSort.

6. You decided to improve insertion sort by using binary search to find the position p where the new insertion should take place:

```
Algorithm ImprovedInsertionSort
    Input/output:  an integer array $a = \{a[0], \ldots, a[n-1]\}$ of size $n$
begin ImprovedInsertionSort
1       for $i \leftarrow 1$ while $i < n$ step $i \leftarrow i + 1$ do
2           $s_\text{tmp} \leftarrow a[i]$; $p \leftarrow$ BinarySearch$(a, i-1, s_\text{tmp})$
3           for integer $j \leftarrow i - 1$ while $j \geq p$ step $j \leftarrow j - 1$ do
4               $a[j+1] \leftarrow a[j]$
5           end for
6           $a[p] \leftarrow s_\text{tmp}$
7       end for
end ImprovedInsertionSort
```

   Here, $s_\text{tmp} = a[i]$ is the current value to be inserted at each step i into the already sorted part $a[0], \ldots, a[i-1]$ of the array a. The binary search along that part returns the position p such that $s_{\text{tmp}} < a[p]$ and either $p = 0$ or $s_{\text{tmp}} \geq a[p-1]$. After finding p, the data values in the subsequent positions $j = i-1, \ldots, p$ are sequentially moved one position up to $i, \ldots, p{+}1$ so that the value s_{tmp} can be inserted into the proper position p.

Answer the following questions regarding time complexity of the proposed algorithm. *Hint*: to answer the questions you may need the formulae for the sum of integers from 1 to n: $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$ and for the sum of their logarithms by the base 2: $\log_2 1 + \log_2 2 + \ldots + \log_2 n \equiv \log_2 n! \geq n \log_2 n - 1.44n$ (see also Appendix **??**).

(a) Determine how many data moves in total should be done in average to sort an array a of the size n.

(b) Determine how many data moves in total should be done in the worst case to sort the array a.

(c) Determine how many comparisons should be done by BinarySearch in average to sort the array a.

(d) Determine how many comparisons should be done by BinarySearch in the worst case to sort the array a.

(e) What is the worst-case complexity of ImprovedInsertionSort if you take account of only the comparisons made by BinarySearch?

(f) What is the worst-case complexity of ImprovedInsertionSort if only moves of the data values are taken into account?

7. Find the best way for selecting p most popular persons in an unordered database containing entries for n different persons all over the world. Each entry i of the data base has a non-negative integer key k_i that numerically specifies how popular is that person. For instance, the key that is less than all other keys characterises the least popular person, and the key that is greater than all other keys belongs to the top-rank "star". You have two options:

   (*i*) repeat p times a sequential search through the whole data base and select each time the top-most key while skipping all the already found keys or

   (*ii*) sort first the whole data base using the QuickSort method and then select the p entries with the top-most keys.

   You found the sequential search and QuickSort have processing times $T(n) = 0.1n$ ms and $T(n) = 0.1n \log_{10} n$ ms, respectively, and the time for selecting an entry from a sorted data base is negligibly small with respect to the sorting time.

   Which option, (*i*) or (*ii*), would you recommend for any particular pair of the values of n and p in order to minimise running time? If your data base contains 10^6 entries and you have to select 100 most popular persons, then which option will be the best?

8. Specify the worst-case and average-case Big-Oh complexity of the following algorithms, assuming an input array of size N: InsertionSort, HeapSort, MergeSort, QuickSort, SequentialSearch, BinarySearch.

9. Convert an array $[10, 26, 52, 76, 13, 8, 3, 33, 60, 42]$ into a maximum heap.

10. Convert the above array into a well-balanced binary search tree. Explain what is the basic difference between a heap and a binary search tree?

11. You have to split an unsorted integer array, W_n, of size n into the two subarrays, U_k and V_{n-k}, of sizes k and $n-k$, respectively, such that all the values in the subarray U_k are smaller than the values in the subarray V_{n-k} . You may use the following two algorithms:

    A. Sort the array W_n with QuickSort and then simply fetch the smallest k items from the positions $i = 0, 1, \ldots, k-1$ of the sorted array to form U_k (you should ignore time of data fetching comparing to the sorting time).

    B. Directly select k smallest items with QuickSelect from the unsorted array (that is, use QuickSelect k times to find items that might be placed to the positions $i = 0, 1, \ldots, k-1$ of the sorted array).

    Let processing time of QuickSort and QuickSelect be $T_{\text{sort}}(n) = c_{\text{sort}} n \log_2 n$ and $T_{\text{slct}}(n) = c_{\text{slct}} n$, respectively. Choose the algorithm with the smallest processing time for any particular n and k.

    Assume that in the previous case QuickSort spends 10.24 microseconds to sort $n = 1024$ unsorted items and QuickSelect spends 1.024 microseconds to select a single i-smallest item from $n = 1024$ unsorted items. What algorithm, **A** or **B**, must be chosen if $n = 2^{20}$ and $k = 2^9 \equiv 512$?

2.2 Sorting and searching: solutions

1. The scaling factors for the linear search and the sort are $\frac{0.1}{100} = 0.001$ and $\frac{0.1}{100 \log 100} = \frac{1}{200} = 0.0005$, respectively (you may use any convenient base of the logarithm for computing the running time). Then the overall time for the scheme **A** is $T_{\text{A}} = 0.001 \cdot 10^7 \cdot 10^3 = 10^7$ ms, or 10^4 seconds. The overall time for **B** consists of the time to convert the list into an array: $T_{\text{B},1} = 0.001 \cdot 10^7$ ms, or 10 seconds, and the time for sorting the array:

$$T_{\text{B},2} = 0.0005 \cdot 10^7 \cdot \log(10^7) = 35 \cdot 10^3 \text{ms} = 35\text{s}.$$

   Therefore, the scheme of choice is **B**.

2. Let an array of size n be split into k subarrays of size $\frac{n}{k}$. Then the total time to separately sort the subarrays is $k \cdot c \cdot \left(\frac{n}{k}\right)^2 = c \cdot \frac{n^2}{k}$. Time for merging k presorted subarrays is $c \cdot (k-1) \cdot n$. Thus the total time is proportional to $\frac{n^2}{k} + n \cdot (k-1)$.

   If $k = 1$ the sorting is $O(n^2)$. If $k = n$ the sorting is also $O(n^2)$. This suggests that the computational complexity may be less somewhere in between these bounds.

Those who know the calculus may easily derive the optimum k by differentiating the expression $\frac{n}{k} + k - 1$ by k and setting it equal to zero: $-\frac{n}{k^2} + 1 = 0$, or $k = \sqrt{n}$. Those who do not like mathematics may exploit the inverse symmetry between the terms $\frac{n}{k}$ and k in this sum. These terms have the fixed product $n = \frac{n}{k} \cdot k$, so they should be set equal for the optimum point: $\frac{n}{k} = k$, or $k = \sqrt{n}$. Both variants of getting the result are acceptable.

The resulting time complexity of this sorting is $O(n^{1.5})$ so that the algorithm is better than InsertionSort but worse than MergeSort.

3. The implicit formula $T(3^m) = 3T(3^{m-1}) + 2c3^m$ can be reduced to

$$\frac{T(3^m)}{3^m} = \frac{T(3^{m-1})}{3^{m-1}} + 2c$$

thus by telescoping:

$$\begin{array}{lcl} \frac{T(3^m)}{3^m} & = & \frac{T(3^{m-1})}{3^{m-1}} + 2c \\ \frac{T(3^{m-1})}{3^{m-1}} & = & \frac{T(3^{m-2})}{3^{m-2}} + 2c \\ \cdots & & \cdots \\ \frac{T(3^1)}{3^1} & = & \frac{T(3^0)}{3^0} + 2c \\ T(1) & = & 0 \end{array}$$

so that $\frac{T(3^m)}{3^m} = 2cm$, or $T(n) = 2cn \log_3 n$

Another solution: by math induction

$$\begin{array}{lclcl} T(3^1) & = & 3 \cdot T(1) + 2 \cdot 3c & = & 2 \cdot 3c \\ T(3^2) & = & 2 \cdot 3^2 c + 2 * 3^2 c & = & 2 \cdot 2 \cdot 3^2 c \\ T(3^3) & = & 2 \cdot 2 \cdot 3^3 c + 2 \cdot 3^3 c & = & 2 \cdot 3 \cdot 3^3 c \end{array}$$

Let us assume that $T(3^k) = 2ck \cdot 3^k$ holds for $k = 1, \dots, m-1$. Then

$$T(3^m) = 3 \cdot 2 \cdot (m-1) \cdot 3^{m-1} c + 2 \cdot 3^m c = 2 \cdot m \cdot 3^m c$$

so that the assumption is valid by induction for all values m. The "Big-Oh" complexity of this algorithm is $O(n \log n)$.

4. For Mergesort, $T(n) = cn \log_2 n$, therefore, to compare the algorithms one must derive whether $cn \log_2 n$ is greater or lesser than $2cn \log_3 n$, that is, whether $\log_2 n$ is greater or lesser than $2 \log_3 n = 2 \log_3 2 \cdot \log_2 n$. Because $2 \log_3 2 = 1.262 > 1.0$, the algorithm under consideration is slightly slower than Mergesort.

5. Let an array of size n be split into k subarrays of size $\frac{n}{k}$. Then the total time to separately sort the subarrays is $k \cdot c \cdot \left(\frac{n}{k}\right)^2 = c \cdot \frac{n^2}{k}$. Time for merging k presorted subarrays is $c \cdot (k-1) \cdot n$. Thus the total time is proportional to $\frac{n^2}{k} + n \cdot (k-1)$.

If $k = 1$ the sorting is $O(n^2)$. If $k = n$ the sorting is also $O(n^2)$. This suggests that the computational complexity may be less somewhere in between these bounds.

Those who know the calculus may easily derive the optimum k by differentiating the expression $\frac{n}{k} + k - 1$ by k and setting it equal to zero: $-\frac{n}{k^2} + 1 = 0$, or $k = \sqrt{n}$. Those who do not like mathematics may exploit the inverse symmetry between the terms $\frac{n}{k}$ and k in this sum. These terms have the fixed product $n = \frac{n}{k} \cdot k$, so they should be set equal for the optimum point: $\frac{n}{k} = k$, or $k = \sqrt{n}$. Both variants of getting the result are acceptable.

The resulting time complexity of this sorting is $O(n^{1.5})$ so that the algorithm is better than InsertionSort but worse than MergeSort.

6. Time complexity of the proposed algorithm depends both on data moves and data comparisons for searching a position of every item.

   (a) Because the position of inserting the i-th item may equiprobably be from 0 to i, in average about $i/2$ data moves should be done. In total:

   $$\frac{1}{2}(0 + 1 + \ldots + i) = \frac{n(n+1)}{4}$$

   (b) Twice more: $\frac{n(n+1)}{2}$ moves in the worst case

   (c) The binary search in the sorted part of i items takes $\frac{1}{2}\log_2 i$ steps in the average. Thus in total the average number of comparisons is

   $$\frac{1}{2}[\log_2 1 + \log_2 2 + ... + \log_2 n] = \frac{1}{2}\log_2(n!) \approx \frac{n}{2}\log_2(n)$$

   (d) Twice more, or $n \log_2 n$ comparisons in the worst case

   (e) With respect to comparisons only, it has complexity $O(n \log n)$

   (f) With respect to moves, it is still $O(n^2)$

7. Time for the options (i) and (ii) is $T(n) = cpn$ $T(n) = cn \log_{10} n$, respectively. Thus, the second option is better when $p > \log_{10} n$. To minimise average running time, one has to choose the first option if $log_{10} n \leq p$ and the second option otherwise. Because $log_{10} 10^6 = 6 < 100$, the second option is the best.

8.

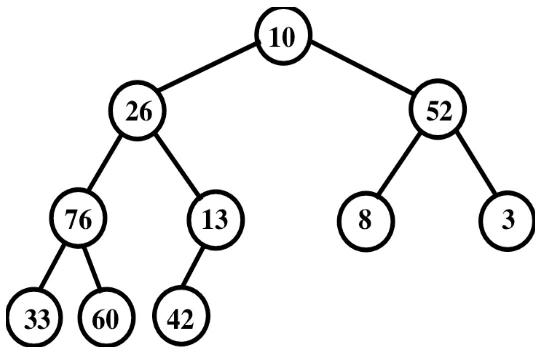
Algorithm	*Worst-case*	*Average-case*
Insertion sort	$O(n^2)$	$O(n^2)$
Heapsort	$O(n \log n)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$
Sequential search	$O(n)$	$O(n)$
Binary search	$O(\log n)$	$O(\log n)$

9. The maximum heap is created from a given array of n integer keys by percolating the keys down starting from the largest non-leaf position $p = \lfloor n/2 \rfloor$ (abbreviation PD:k stands for the percolation down starting from the position k):

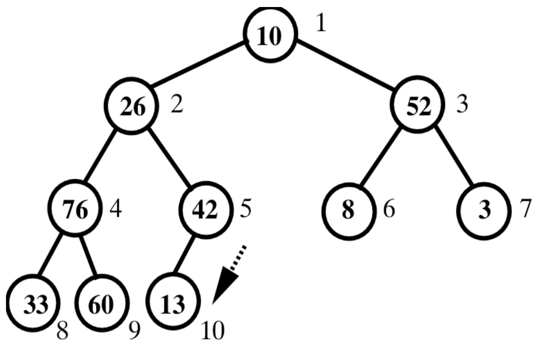
Position	1	2	3	4	5	6	7	8	9	10
Array	10	26	52	76	13	8	3	33	60	42
PD:5	10	26	52	76	**42**	8	3	33	60	13
PD:4	10	26	52	76	42	8	3	33	60	13
PD:3	10	26	52	76	42	8	3	33	60	13
PD:2	10	**76**	52	**26**	42	8	3	33	60	13
	10	76	52	**60**	42	8	3	33	**26**	13
PD:1	**76**	**10**	52	60	42	8	3	33	26	13
	76	**60**	52	**10**	42	8	3	33	26	13
	76	60	52	**33**	42	8	3	**10**	26	13
Max. heap	76	60	52	33	42	8	3	10	26	13

The initial array and the percolation steps of creating the maximum heap are shown in figures below.

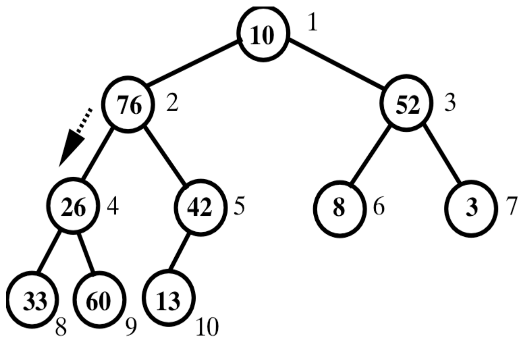
Initial array:

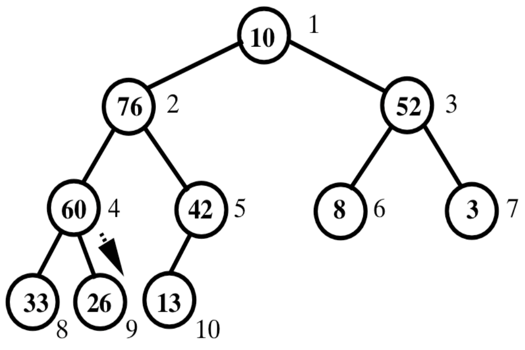


Percolation down from the position 5 (PD:5):

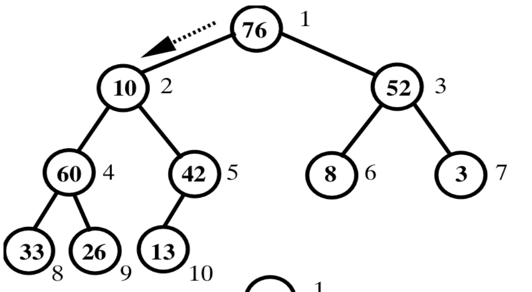


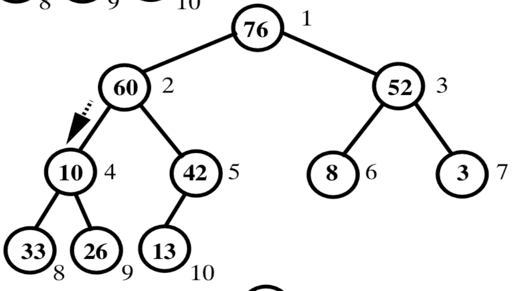
Percolation down from the position 2 (PD:2):

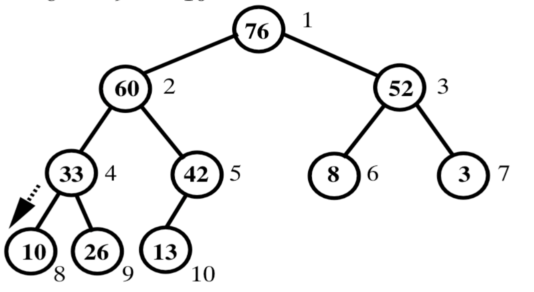




Percolation down from the position 1 (PD:1):



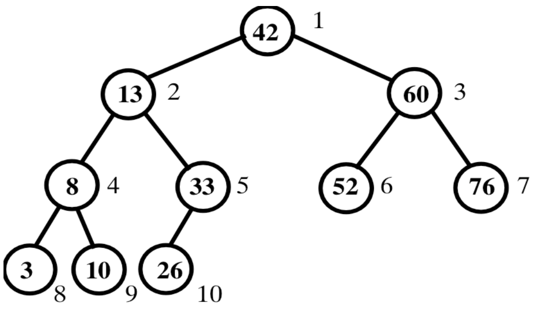




10. The depth-first traversal of a binary search tree puts all the items attached to the nodes into sorted order. Therefore, to convert the array into such a tree, one should sort the array, and place the sorted elements in line with the traversal of the tree. In our case, the depth-first traversal from left to right gives the following sequence of the nodes relating to the sorted array:

Node :	8	4	9	2	5	10	1	6	3	7
Element :	3	8	10	13	33	26	42	52	60	76

so that the well-balanced binary search tree similar to the heap is as follows:



A *binary search tree* satisfies the following search order priority: for every node x in the tree, the values of all the keys in the left subtree are smaller than or equal to the key in x and the values of all the keys in the right subtree are greater than the key in x. In a *heap*, the key of each parent node is greater than or equal to the key of any child node.

11. The algorithms **A** and **B** spend time $T_{\mathrm{A}} = c_{\mathrm{sort}} n \log_2 n$ and $T_{\mathrm{B}} = c_{\mathrm{slct}} kn$, respectively, to select the desired k items. Therefore, the algorithm **B** outperforms the algorithm **A** if $c_{\mathrm{slct}} kn \le c_{\mathrm{sort}} n \log_2 n$, i.e. if $k \le \frac{c_{\mathrm{sort}}}{c_{\mathrm{slct}}} \log_2 n$.

    In accord with the given conditions, $c_{\mathrm{sort}} 1024 \log_2 1024 = 10240 c_{\mathrm{sort}} =$ 10.24 microseconds for QuickSort, so that $c_{\mathrm{sort}} = 10^{-3}$ microseconds per item. For QuickSelect, $c_{\mathrm{slct}} 1024 = 1.024$microseconds, so that $c_{\mathrm{slct}} = 10^{-3}$ microseconds per item. Because $k = 512 \gg \frac{10^{-3}}{10^{-3}} \log_2 2^{20} = 20.$ the algorithm of choice for $k = 512$ and $n = 2^{20}$ is **A**.

2.3 Symbol tables and hashing: exercises

1. The NZ Inland Revenue Department assigns a unique 8-digit decimal IRD number in the range [00,000,000 ... 99,999,999] to every person paying taxes. Provided that the amount of random access memory (RAM) available is not restricted, you can sort an arbitrary large database of the IRD numbers using an integer array of size 100,000,000. Each element of the array is a counter that counts how many times the IRD number occurs in the database (zero if the IRD number is absent).

    (a) What is the Big-Oh complexity of this algorithm, which is called a **bucket sort**, for sorting a database of size N?

    (b) What is the Big-Oh complexity of checking whether a given IRD number is present or absent in the sorted database?

    (c) What is the Big-Oh complexity of updating the database by deleting or inserting a particular number?

    (d) Explain, why the bucket sort cannot serve as a general sorting algorithm?

2. Convert an array $[10, 26, 52, 76, 13, 8, 3, 33, 60, 42]$ into a hash table of size 13 using modulo-based hash addressing: $\langle\mathsf{address}\rangle = \langle\mathsf{data\ value}\rangle \bmod 13$ and linear probing to resolve collisions. *Hint*: Modulo operation $m \bmod n$ returns the residual of the integer division m/n: e.g., if $m = 17$, $n = 13$ then $17/13 = 1$ and $17 \mathrm{mod}\ 13 = 4$. The operation is denoted $m\%n$ in Java.

3. Convert the above array into a hash table of size 13 using modulo-based hash addressing: $\langle\mathsf{address}\rangle = \langle\mathsf{data\ value}\rangle \mathrm{mod}\ 13$ and double hashing with the backward step $\max\{1, \langle\mathsf{data\ value}\rangle/13\}$ to resolve collisions.

4. Convert the above array into a hash table of size 13 using modulo-based hash addressing $\langle\mathsf{address}\rangle = \langle\mathsf{data\ value}\rangle \mathrm{mod}\ 13$ and chaining to resolve collisions.

2.4 Symbol tables and hashing: solutions

1. The bucket sort has the following properties.

   (a) It has complexity $O(N)$ for sorting an arbitrary database of size N.

   (b) Complexity of checking whether a given IRD number is present or absent in the sorted database is $O(1)$.

   (c) Complexity of updating the database by deleting or inserting a particular number is $O(1)$.

   (d) The bucket sort cannot serve as a general sorting algorithm because in the general case the range of data to be sorted is not restricted.

2. The initially empty hash table of size 13 is formed using the hash address $h(k) = k \bmod 13$ and linear probing (with the probe decrement $\Delta = -1$) as follows:

Key k	Address $h(k)$	Collision resolution
10	10	no collision
26	0	no collision
52	0	collision; probing: $(0-1) \bmod 13 = 12$
	12	no collision
76	11	no collision
13	0	collision; probing: $(0-1) \bmod 13 = 12$
	12	collision; probing: $12-1=11$
	11	collision; probing: $11-1=10$
	10	collision; probing: $10-1=9$
	9	no collision
8	8	no collision
3	3	no collision
33	7	no collision
60	8	collision; probing: $8-1=7$
	7	collision; probing: $7-1=6$
	6	no collision
42	3	collision; probing: $3-1=2$
	2	no collision

   The final hash array is

address	0	1	2	3	4	5	6	7	8	9	10	11	12
key	26	–	42	3	–	–	60	33	8	13	10	76	52

3. The initially empty hash table of size 13 is formed using the hash address $h(k) = k \bmod 13$ and double hashing probing (with the probe decrement $\Delta(k) = \max\{k/13,\ 1\}$) as follows:

Key k	Address $h(k)$	$\Delta(k)$	Collision resolution
10	10		no collision
26	0		no collision
52	0 9	4	collision; probing: $(0-4)\text{mod } 13 = 9$ no collision
76	11		no collision
13	0 12	1	collision; probing $(0-1)\text{mod } 13 = 12$ no collision
8	8		no collision
3	3		no collision
33	7		no collision
60	8 4	4	collision; probing $(8-4) = 4$ no collision
42	3 0 10 7 4 1	3	collision; probing $(3-3) = 0$ collision; probing $(0-3)\text{mod } 13 = 10$ collision; probing $(10-3) = 7$ collision; probing $(7-3) = 4$ collision; probing $(4-3) = 1$ no collision

The final hash array is

address	0	1	2	3	4	5	6	7	8	9	10	11	12
key	26	42	–	3	60	–	–	33	8	52	10	76	13

4. The initially empty hash table of size 13 is formed using the hash address $h(k) = k\text{mod } 13$ and external chaining as follows:

Key k	10	26	52	76	13	8	3	33	60	42
$h(k)$	10	0	0	11	0	8	3	7	8	3
Chain	10	26	26 52	76	26 52 13	8	3	33	8 60	3 42

The final hash array with the chains of keys is

address	0	1	2	3	4	5	6	7	8	9	10	11	12
chain	26 52 13	–	–	3 42	–	–	–	33	8 60	–	10	76	–