Chapter 2

Data Structures and Libraries

If I have seen further it is only by standing on the shoulders of giants.
— **Isaac Newton**

2.1 Overview and Motivation

A data structure (DS) is a means of storing and organizing data. Different data structures have different strengths. So when designing an algorithm, it is important to pick one that allows for efficient insertions, searches, deletions, queries, and/or updates, depending on what your algorithm needs. Although a data structure does not in itself solve a (programming contest) problem (the algorithm operating on it does), using an appropriately efficient data structure for a problem may be the difference between passing or exceeding the problem's time limit. There can be many ways to organize the same data and sometimes one way is better than the other in some contexts. We will illustrate this several times in this chapter. A keen familiarity with the data structures and libraries discussed in this chapter is critically important for understanding the algorithms that use them in subsequent chapters.

As stated in the preface of this book, we **assume** that you are *familiar* with the basic data structures listed in Section 2.2-2.3 and thus we will **not** review them in this book. Instead, we will simply highlight the fact that there exist built-in implementations for these elementary data structures in the C++ STL and Java API[1]. If you feel that you are not entirely familiar with any of the terms or data structures mentioned in Section 2.2-2.3, please review those particular terms and concepts in the various reference books[2] that cover them, including classics such as the "Introduction to Algorithms" [7], "Data Abstraction and Problem Solving" [5, 54], "Data Structures and Algorithms" [12], etc. Continue reading this book only when you understand at least the *basic concepts* behind these data structures.

Note that for competitive programming, you only need to know enough about these data structures to be able to select and to *use* the correct data structures for each given contest problem. You should understand the strengths, weaknesses, and time/space complexities of typical data structures. The theory behind them is definitely good reading, but can often be skipped or skimmed through, since the built-in libraries provide ready-to-use and highly reliable implementations of otherwise complex data structures. This is *not* a good practice, but you will find that it is often sufficient. Many (younger) contestants have been able to utilize the efficient (with a $O(\log n)$ complexity for most operations) C++ STL `map` (or

[1]Even in this third edition, we *still* primarily use C++ code to illustrate implementation techniques. The Java equivalents can be found in the supporting website of this book.

[2]Materials in Section 2.2-2.3 are usually covered in year one *Data Structures* CS curriculae. High school students aspiring to take part in the IOI are encouraged to engage in independent study on such material.

Java `TreeMap`) implementations to store dynamic collections of key-data pairs without an understanding that the underlying data structure is a *balanced Binary Search Tree*, or use the C++ STL `priority_queue` (or Java `PriorityQueue`) to order a queue of items without understanding that the underlying data structure is a *(usually Binary) Heap*. Both data structures are typically taught in year one Computer Science curriculae.

This chapter is divided into three parts. Section 2.2 contains basic *linear* data structures and the basic operations they support. Section 2.3 covers basic *non-linear* data structures such as (balanced) Binary Search Trees (BST), (Binary) Heaps, and Hash Tables, as well as their basic operations. The discussion of each data structure in Section 2.2-2.3 is brief, with an emphasis on the important *library routines* that exist for manipulating the data structures. However, special data structures that are common in programming contests, such as bitmask and several bit manipulation techniques (see Figure 2.1) are discussed in more detail. Section 2.4 contains *more* data structures for which there exist no built-in implementation, and thus require us to build *our own* libraries. Section 2.4 has a more in-depth discussion than Section 2.2-2.3.

Value-Added Features of this Book

As this chapter is the first that dives into the heart of competitive programming, we will now take the opportunity to highlight several value-added features of this book that you will see in this and the following chapters.

A key feature of this book is its accompanying collection of *efficient, fully-implemented examples* in both C/C++ and Java that many other Computer Science books lack, stopping at the 'pseudo-code level' in their demonstration of data structures and algorithms. This feature has been in the book since the very first edition. The important parts of the source code have been included in the book[3] and the full source code is hosted at `sites.google.com/site/stevenhalim/home/material`. The reference to each source file is indicated in the body text as a box like the one shown below.

Source code: `chx_yy_name.cpp/java`

Another strength of this book is the collection of both written and programming exercises (mostly supported by the UVa Online Judge [47] and integrated with uHunt—see Appendix A). In the *third* edition, we have added *many more* written exercises. We have classified the written exercises into *non-starred* and *starred* ones. The non-starred written exercises are meant to be used mainly for self-checking purposes; solutions are given at the back of each chapter. The starred written exercises can be used for extra challenges; we do not provide solutions for these but may instead provide some helpful hints.

In the *third* edition, we have added visualizations[4] for many data structures and algorithms covered in this book [27]. We believe that these visualizations will be a huge benefit to the visual learners in our reader base. At this point in time (24 May 2013), the visualizations are hosted at: `www.comp.nus.edu.sg/~stevenha/visualization`. The reference to each visualization is included in the body text as a box like the one shown below.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization`

[3]However, we have chosen not to include code from Section 2.2-2.3 in the body text because they are mostly 'trivial' for many readers, except perhaps for a few useful tricks.

[4]They are built with HTML5 canvas and JavaScript technology.

2.2 Linear DS with Built-in Libraries

A data structure is classified as a *linear* data structure if its elements form a linear sequence, i.e. its elements are arranged from left to right (or top to bottom). Mastery of these basic linear data structures below is critical in today's programming contests.

- Static Array (native support in both C/C++ and Java)
  This is clearly the most commonly used data structure in programming contests. Whenever there is a collection of sequential data to be stored and later accessed using their *indices*, the static array is the most natural data structure to use. As the maximum input size is usually mentioned in the problem statement, the array size can be declared to be the maximum input size, with a small extra buffer (sentinel) for safety—to avoid the unnecessary 'off by one' RTE. Typically, 1D, 2D, and 3D arrays are used in programming contests—problems rarely require arrays of higher dimension. Typical array operations include accessing elements by their indices, sorting elements, performing a linear scan or a binary search on a sorted array.

- Dynamically-Resizeable Array: C++ STL `vector` (Java `ArrayList` (faster) or `Vector`)
  This data structure is similar to the static array, except that it is designed to handle runtime resizing natively. It is better to use a `vector` in place of an array if the size of the sequence of elements is unknown at compile-time. Usually, we initialize the size (`reserve()` or `resize()`) with the estimated size of the collection for better performance. Typical C++ STL `vector` operations used in competitive programming include `push_back()`, `at()`, the `[]` operator, `assign()`, `clear()`, `erase()`, and `iterator`s for traversing the contents of `vector`s.

  Source code: `ch2_01_array_vector.cpp/java`

  It is appropriate to discuss two operations commonly performed on Arrays: **Sorting** and **Searching**. These two operations are well supported in C++ and Java.

  There are *many* sorting algorithms mentioned in CS books [7, 5, 54, 12, 40, 58], e.g.

  1. $O(n^2)$ comparison-based sorting algorithms: Bubble/Selection/Insertion Sort, etc. These algorithms are (awfully) slow and usually avoided in programming contests, though understanding them might help you solve certain problems.
  2. $O(n \log n)$ comparison-based sorting algorithms: Merge/Heap/Quick Sort, etc. These algorithms are the default choice in programming contests as an $O(n \log n)$ complexity is optimal for comparison-based sorting. Therefore, these sorting algorithms run in the 'best possible' time in most cases (see below for special purpose sorting algorithms). In addition, these algorithms are well-known and hence we do not need to 'reinvent the wheel'[5]—we can simply use `sort`, `partial_sort`, or `stable_sort` in C++ STL `algorithm` (or `Collections.sort` in Java) for standard sorting tasks. We only need to specify the required comparison function and these library routines will handle the rest.
  3. Special purpose sorting algorithms: $O(n)$ Counting/Radix/Bucket Sort, etc. Although rarely used, these special purpose algorithms are good to know as they can reduce the required sorting time if the data has certain special characteristics. For example, Counting Sort can be applied to integer data that lies in a small range (see Section 9.32).

[5]However, sometimes we do need to 'reinvent the wheel' for certain sorting-related problems, e.g. the Inversion Index problem in Section 9.14.

There are generally three common methods to search for an item in an array:

1. $O(n)$ Linear Search: Consider every element from index 0 to index $n-1$ (avoid this whenever possible).
2. $O(\log n)$ Binary Search: Use `lower_bound`, `upper_bound`, or `binary_search` in C++ STL `algorithm` (or Java `Collections.binarySearch`). If the input array is unsorted, it is necessary to sort the array at least once (using one of the $O(n \log n)$ sorting algorithm above) before executing one (or *many*) Binary Search(es).
3. $O(1)$ with Hashing: This is a useful technique to use when fast access to known values are required. If a suitable hash function is selected, the probability of a collision to be made is negligibly small. Still, this technique is rarely used and we can live without it[6] for most (contest) problems.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/sorting.html`

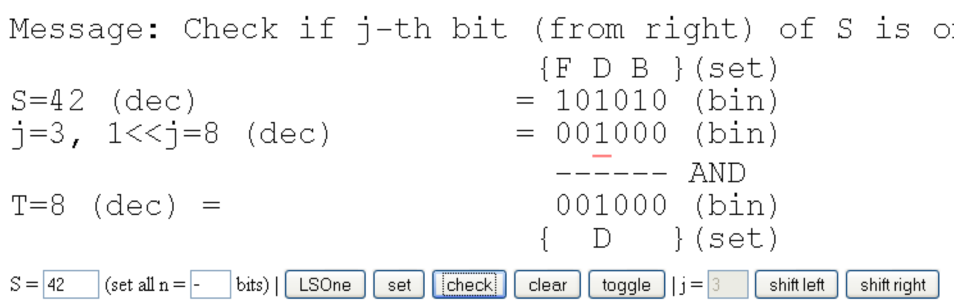
Source code: `ch2_02_algorithm_collections.cpp/java`

- Array of Booleans: C++ STL `bitset` (Java `BitSet`)

  If our array needs only to contain Boolean values (1/true and 0/false), we can use an alternative data structure other than an array—a C++ STL `bitset`. The `bitset` supports useful operations like `reset()`, `set()`, the `[]` operator and `test()`.

  Source code: `ch5_06_primes.cpp/java`, also see Section 5.5.1

- Bitmasks a.k.a. lightweight, small sets of Booleans (native support in C/C++/Java)
  An integer is stored in a computer's memory as a sequence/string of bits. Thus, we can use integers to represent a *lightweight* small set of Boolean values. All set operations then involve only the bitwise manipulation of the corresponding integer, which makes it a *much more efficient* choice when compared with the C++ STL `vector<bool>`, `bitset`, or `set<int>` options. Such speed is important in competitive programming. *Some* important operations that are used in this book are shown below.



```
Message: Check if j-th bit (from right) of S is on
                                 {F D B }(set)
S=42 (dec)                     = 101010 (bin)
j=3, 1<<j=8 (dec)              = 001000 (bin)
                                 ------ AND
T=8 (dec) =                      001000 (bin)
                                 {  D   }(set)
```

Figure 2.1: Bitmask Visualization

1. Representation: A 32 (or 64)-bit *signed* integer for up to 32 (or 64) items[7]. Without a loss of generality, all examples below use a 32-bit signed integer called S.

[6]However, questions about hashing frequently appear in interviews for IT jobs.

[7]To avoid issues with the two's complement representation, use a 32-bit/64-bit *signed* integer to represent bitmasks of up to 30/62 items only, respectively.

```
Example:            5| 4| 3| 2| 1| 0 <- 0-based indexing from right
                   32|16| 8| 4| 2| 1 <- power of 2
S = 34 (base 10) =  1| 0| 0| 0| 1| 0 (base 2)
                    F| E| D| C| B| A <- alternative alphabet label
```

In the example above, the integer $S = 34$ or `100010` in binary also represents a small set {1, 5} with a 0-based indexing scheme in increasing digit significance (or {B, F} using the alternative alphabet label) because the second and the sixth bits (counting from the right) of S are on.

2. To multiply/divide an integer by 2, we only need to shift the bits in the integer left/right, respectively. This operation (especially the shift left operation) is important for the next few examples below. Notice that the truncation in the shift right operation automatically rounds the division-by-2 down, e.g. $17/2 = 8$.

```
S                  = 34 (base 10) =  100010 (base 2)
S = S << 1 = S * 2 = 68 (base 10) = 1000100 (base 2)
S = S >> 2 = S / 4 = 17 (base 10) =   10001 (base 2)
S = S >> 1 = S / 2 =  8 (base 10) =    1000 (base 2) <- LSB is gone
                                          (LSB = Least Significant Bit)
```

3. To set/turn on the `j`-th item (0-based indexing) of the set, use the bitwise OR operation `S |= (1 << j)`.

```
S = 34 (base 10) = 100010 (base 2)
j = 3, 1 << j    = 001000 <- bit '1' is shifted to the left 3 times
                   -------- OR (true if either of the bits is true)
S = 42 (base 10) = 101010 (base 2) // update S to this new value 42
```

4. To check if the `j`-th item of the set is on, use the bitwise AND operation `T = S & (1 << j)`.
If `T = 0`, then the `j`-th item of the set is off.
If `T != 0` (to be precise, `T = (1 << j)`), then the `j`-th item of the set is on.
See Figure 2.1 for one such example.

```
S = 42 (base 10) = 101010 (base 2)
j = 3, 1 << j    = 001000 <- bit '1' is shifted to the left 3 times
                   -------- AND (only true if both bits are true)
T = 8 (base 10)  = 001000 (base 2) -> not zero, the 3rd item is on

S = 42 (base 10) = 101010 (base 2)
j = 2, 1 << j    = 000100 <- bit '1' is shifted to the left 2 times
                   -------- AND
T = 0 (base 10)  = 000000 (base 2) -> zero, the 2rd item is off
```

5. To clear/turn off the `j`-th item of the set, use[8] the bitwise AND operation `S &= ~(1 << j)`.

```
S = 42 (base 10) = 101010 (base 2)
j = 1, ~(1 << j) = 111101 <- '~' is the bitwise NOT operation
                   -------- AND
S = 40 (base 10) = 101000 (base 2) // update S to this new value 40
```

[8]Use brackets a lot when doing bit manipulation to avoid accidental bugs due to operator precedence.

6. To toggle (flip the status of) the `j`-th item of the set, use the bitwise XOR operation `S ^= (1 << j)`.

```
S = 40 (base 10) = 101000 (base 2)
j = 2, (1 << j)  = 000100 <- bit '1' is shifted to the left 2 times
                   -------- XOR <- true if both bits are different
S = 44 (base 10) = 101100 (base 2) // update S to this new value 44

S = 40 (base 10) = 101000 (base 2)
j = 3, (1 << j)  = 001000 <- bit '1' is shifted to the left 3 times
                   -------- XOR <- true if both bits are different
S = 32 (base 10) = 100000 (base 2) // update S to this new value 32
```

7. To get the value of the least significant bit that is on (first from the right), use `T = (S & (-S))`.

```
 S   =  40 (base 10) =  000...000101000 (32 bits, base 2)
-S   = -40 (base 10) =  111...111011000 (two's complement)
                       ----------------- AND
 T   =   8 (base 10) =  000...000001000 (3rd bit from right is on)
```

8. To turn on *all* bits in a set of size n, use `S = (1 << n) - 1` (be careful with overflows).

```
Example for n = 3
S + 1 = 8 (base 10) = 1000 <- bit '1' is shifted to left 3 times
                         1
                      ------ -
S     = 7 (base 10) =  111 (base 2)

Example for n = 5
S + 1 = 32 (base 10) = 100000 <- bit '1' is shifted to left 5 times
                            1
                       -------- -
S     = 31 (base 10) =  11111 (base 2)
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bitmask.html`

Source code: `ch2_03_bit_manipulation.cpp/java`

Many bit manipulation operations are written as preprocessor macros in our C/C++ example source code (but written plainly in our Java example code since Java does not support macros).

- Linked List: C++ STL `list` (Java `LinkedList`)
  Although this data structure almost always appears in data structure and algorithm textbooks, the Linked List is usually avoided in typical (contest) problems. This is due to the inefficiency in accessing elements (a linear scan has to be performed from the head or the tail of a list) and the usage of pointers makes it prone to runtime errors if not implemented properly. In this book, almost all forms of Linked List have been replaced by the more flexible C++ STL `vector` (Java `Vector`).

The only exception is probably UVa 11988 - Broken Keyboard (a.k.a. Beiju Text)—where you are required to dynamically maintain a (linked) list of characters and efficiently insert a new character *anywhere* in the list, i.e. at front (head), current, or back (tail) of the (linked) list. Out of 1903 UVa problems that the authors have solved, this is likely to be the only pure linked list problem we have encountered thus far.

- Stack: C++ STL `stack` (Java `Stack`)
  This data structure is often used as part of algorithms that solve certain problems (e.g. bracket matching in Section 9.4, Postfix calculator and Infix to Postfix conversion in Section 9.27, finding Strongly Connected Components in Section 4.2.9 and Graham's scan in Section 7.3.7). A stack only allows for $O(1)$ insertion (push) and $O(1)$ deletion (pop) from the top. This behavior is usually referred to as Last In First Out (LIFO) and is reminiscent of literal stacks in the real world. Typical C++ STL `stack` operations include `push()/pop()` (insert/remove from top of stack), `top()` (obtain content from the top of stack), and `empty()`.

- Queue: C++ STL `queue` (Java `Queue`[9])
  This data structure is used in algorithms like Breadth First Search (BFS) in Section 4.2.2. A queue only allows for $O(1)$ insertion (enqueue) from the back (tail) and $O(1)$ deletion (dequeue) from the front (head). This behavior is similarly referred to as First In First Out (FIFO), just like actual queues in the real world. Typical C++ STL `queue` operations include `push()/pop()` (insert from back/remove from front of queue), `front()/back()` (obtain content from the front/back of queue), and `empty()`.

- Double-ended Queue (Deque): C++ STL `deque` (Java `Deque`[10])
  This data structure is very similar to the resizeable array (vector) and queue above, except that deques support fast $O(1)$ insertions and deletions at *both* the beginning and the end of the deque. This feature is important in certain algorithm, e.g. the Sliding Window algorithm in Section 9.31. Typical C++ STL `deque` operations include `push_back()`, `pop_front()` (just like the normal queue), `push_front()` and `pop_back()` (specific for deque).

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/list.html`

Source code: `ch2_04_stack_queue.cpp/java`

Exercise 2.2.1*: Suppose you are given an *unsorted* array S of n integers. Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints: $1 \leq n \leq 100K$ so that $O(n^2)$ solutions are theoretically infeasible in a contest environment.

1. Determine if S contains one or more pairs of duplicate integers.

2*. Given an integer v, find two integers $a, b \in S$ such that $a + b = v$.

3*. Follow-up to Question 2: what if the given array S is *already sorted*?

4*. Print the integers in S that fall between a range $[a \ldots b]$ (inclusive) in sorted order.

5*. Determine the length of the longest increasing *contiguous* sub-array in S.

6. Determine the median (50th percentile) of S. Assume that n is odd.

[9]The Java `Queue` is only an *interface* that is usually instantiated with Java `LinkedList`.

[10]The Java `Deque` is also an *interface*. `Deque` is usually instantiated with Java `LinkedList`.

Exercise 2.2.2: There are several other 'cool' tricks possible with bit manipulation techniques but these are rarely used. Please implement these tasks with bit manipulation:

1. Obtain the remainder (modulo) of S when it is divided by N (N is a power of 2) e.g. $S = (7)_{10}$ % $(4)_{10} = (111)_2$ % $(100)_2 = (11)_2 = (3)_{10}$.
2. Determine if S is a power of 2. e.g. $S = (7)_{10} = (111)_2$ is not a power of 2, but $(8)_{10} = (100)_2$ is a power of 2.
3. Turn off the last bit in S, e.g. $S = (40)_{10} = (10\underline{1}000)_2 \rightarrow S = (32)_{10} = (10\underline{0}000)_2$.
4. Turn on the last zero in S, e.g. $S = (41)_{10} = (1010\underline{0}1)_2 \rightarrow S = (43)_{10} = (1010\underline{1}1)_2$.
5. Turn off the last consecutive run of ones in S e.g. $S = (39)_{10} = (100\underline{111})_2 \rightarrow S = (32)_{10} = (100\underline{000})_2$.
6. Turn on the last consecutive run of zeroes in S e.g. $S = (36)_{10} = (1001\underline{00})_2 \rightarrow S = (39)_{10} = (1001\underline{11})_2$.

7*. Solve UVa 11173 - Grey Codes with a *one-liner* bit manipulation expression for each test case, i.e. find the k-th Gray code.

8*. Let's reverse the UVa 11173 problem above. Given a gray code, find its position k using bit manipulation.

Exercise 2.2.3*: We can also use a *resizeable* array (C++ STL `vector` or Java `Vector`) to implement an efficient stack. Figure out how to achieve this. Follow up question: Can we use a *static* array, linked list, or deque instead? Why or why not?

Exercise 2.2.4*: We can use a linked list (C++ STL `list` or Java `LinkedList`) to implement an efficient queue (or deque). Figure out how to achieve this. Follow up question: Can we use a resizeable array instead? Why or why not?

Programming exercises involving linear data structures (and algorithms) with libraries:

- 1D Array Manipulation, e.g. array, C++ STL `vector` (or Java `Vector`/`ArrayList`)
  1. *UVa 00230 - Borrowers* (a bit of string parsing, see Section 6.2; maintain list of sorted books; sort key: author names first and if ties, by title; the input size is small although not stated; we do not need to use balanced BST)
  2. UVa 00394 - Mapmaker (any n-dimensional array is stored in computer memory as a single dimensional array; follow the problem description)
  3. UVa 00414 - Machined Surfaces (get longest stretch of 'B's)
  4. UVa 00467 - Synching Signals (linear scan, 1D boolean flag)
  5. UVa 00482 - Permutation Arrays (you may need to use a string tokenizer—see Section 6.2—as the size of the array is not specified)
  6. UVa 00591 - Box of Bricks (sum all items; get the average; sum the total absolute differences of each item from the average divided by two)
  7. *UVa 00665 - False Coin* (use 1D boolean flags; all coins are initially potential false coins; if '=', all coins on the left and right are not false coins; if '<' or '>', all coins not on the left and right are not false coins; check if there is only one candidate false coin left at the end)
  8. UVa 00755 - 487-3279 (Direct Addressing Table; convert the letters except Q & Z to 2-9; keep '0'-'9' as 0-9; sort the integers; find duplicates if any)

9. **UVa 10038 - Jolly Jumpers *** (use 1D boolean flags to check $[1..n-1]$)
10. UVa 10050 - Hartals (1D boolean flag)
11. UVa 10260 - Soundex (Direct Addressing Table for soundex code mapping)
12. UVa 10978 - Let's Play Magic (1D string array manipulation)
13. *UVa 11093 - Just Finish it up* (linear scan, circular array, a bit challenging)
14. UVa 11192 - Group Reverse (character array)
15. UVa 11222 - Only I did it (use several 1D arrays to simplify this problem)
16. **UVa 11340 - Newspaper *** (DAT; see Hashing in Section 2.3)
17. UVa 11496 - Musical Loop (store data in 1D array, count the peaks)
18. UVa 11608 - No Problem (use three arrays: created; required; available)
19. UVa 11850 - Alaska (for each integer location from 0 to 1322; can Brenda reach (anywhere within 200 miles of) any charging stations?)
20. *UVa 12150 - Pole Position* (simple manipulation)
21. ***UVa 12356 - Army Buddies **** (similar to deletion in doubly linked lists, but we can still use a 1D array for the underlying data structure)

- 2D Array Manipulation

1. UVa 00101 - The Blocks Problem ('stack' like simulation; but we need to access the content of each stack too, so it is better to use 2D array)
2. UVa 00434 - Matty's Blocks (a kind of visibility problem in geometry, solvable with using 2D array manipulation)
3. UVa 00466 - Mirror Mirror (core functions: rotate and reflect)
4. UVa 00541 - Error Correction (count the number of '1's for each row/col; all of them must be even; if $\exists$ an error, check if it is on the same row and col)
5. UVa 10016 - Flip-flop the Squarelotron (tedious)
6. UVa 10703 - Free spots (use 2D boolean array of size 500×500)
7. **UVa 10855 - Rotated squares *** (string array, 90^o clockwise rotation)
8. **UVa 10920 - Spiral Tap *** (simulate the process)
9. UVa 11040 - Add bricks in the wall (non trivial 2D array manipulation)
10. UVa 11349 - Symmetric Matrix (use long long to avoid issues)
11. UVa 11360 - Have Fun with Matrices (do as asked)
12. **UVa 11581 - Grid Successors *** (simulate the process)
13. UVa 11835 - Formula 1 (do as asked)
14. *UVa 12187 - Brothers* (simulate the process)
15. *UVa 12291 - Polyomino Composer* (do as asked, a bit tedious)
16. *UVa 12398 - NumPuzz I* (simulate backwards, do not forget to mod 10)

- C++ STL `algorithm` (Java `Collections`)

1. UVa 00123 - Searching Quickly (modified comparison function, use `sort`)
2. **UVa 00146 - ID Codes *** (use `next_permutation`)
3. UVa 00400 - Unix ls (this command very frequently used in UNIX)
4. UVa 00450 - Little Black Book (tedious sorting problem)
5. *UVa 00790 - Head Judge Headache* (similar to UVa 10258)
6. UVa 00855 - Lunch in Grid City (sort, median)
7. UVa 01209 - Wordfish (LA 3173, Manila06) (STL `next` and `prev_permutation`)
8. *UVa 10057 - A mid-summer night ...* (involves the median, use STL `sort`, `upper_bound`, `lower_bound` and some checks)

9. **UVa 10107 - What is the Median? *** (find median of a *growing*/dynamic list of integers; still solvable with multiple calls of `nth_element` in `algorithm`)
10. UVa 10194 - Football a.k.a. Soccer (multi-fields sorting, use `sort`)
11. **UVa 10258 - Contest Scoreboard *** (multi-fields sorting, use `sort`)
12. *UVa 10698 - Football Sort* (multi-fields sorting, use `sort`)
13. UVa 10880 - Colin and Ryan (use `sort`)
14. UVa 10905 - Children's Game (modified comparison function, use `sort`)
15. UVa 11039 - Building Designing (use `sort` then count different signs)
16. UVa 11321 - Sort Sort and Sort (be careful with negative mod!)
17. UVa 11588 - Image Coding (`sort` simplifies the problem)
18. UVa 11777 - Automate the Grades (`sort` simplifies the problem)
19. UVa 11824 - A Minimum Land Price (`sort` simplifies the problem)
20. *UVa 12541 - Birthdates* (LA6148, HatYai12, `sort`, pick youngest and oldest)

- Bit Manipulation (both C++ STL `bitset` (Java `BitSet`) and bitmask)
  1. UVa 00594 - One Little, Two Little ... (manipulate bit string with `bitset`)
  2. UVa 00700 - Date Bugs (can be solved with `bitset`)
  3. UVa 01241 - Jollybee Tournament (LA 4147, Jakarta08, easy)
  4. ***UVa 10264 - The Most Potent Corner **** (heavy bitmask manipulation)
  5. *UVa 11173 - Grey Codes* (D & C pattern or one liner bit manipulation)
  6. UVa 11760 - Brother Arif, ... (separate row+col checks; use two bitsets)
  7. **UVa 11926 - Multitasking *** (use 1M `bitset` to check if a slot is free)
  8. **UVa 11933 - Splitting Numbers *** (an exercise for bit manipulation)
  9. IOI 2011 - Pigeons (this problem becomes simpler with bit manipulation but the final solution requires much more than that.)
- C++ STL `list` (Java `LinkedList`)
  1. **UVa 11988 - Broken Keyboard ... *** (rare linked list problem)
- C++ STL `stack` (Java `Stack`)
  1. UVa 00127 - "Accordian" Patience (shuffling `stack`)
  2. **UVa 00514 - Rails *** (use `stack` to simulate the process)
  3. **UVa 00732 - Anagram by Stack *** (use `stack` to simulate the process)
  4. ***UVa 01062 - Containers **** (LA 3752, WorldFinals Tokyo07, simulation with `stack`; maximum answer is 26 stacks; $O(n)$ solution exists)
  5. UVa 10858 - Unique Factorization (use `stack` to help solving this problem)

  Also see: implicit `stack`s in recursive function calls and Postfix conversion/evaluation in Section 9.27.
- C++ STL `queue` and `deque` (Java `Queue` and `Deque`)
  1. UVa 00540 - Team Queue (modified 'queue')
  2. **UVa 10172 - The Lonesome Cargo ... *** (use both `queue` and `stack`)
  3. **UVa 10901 - Ferry Loading III *** (simulation with `queue`)
  4. UVa 10935 - Throwing cards away I (simulation with `queue`)
  5. **UVa 11034 - Ferry Loading IV *** (simulation with `queue`)
  6. *UVa 12100 - Printer Queue* (simulation with `queue`)
  7. *UVa 12207 - This is Your Queue* (use both `queue` and `deque`)

  Also see: `queue`s in BFS (see Section 4.2.2)

2.3 Non-Linear DS with Built-in Libraries

For some problems, linear storage is not the best way to organize data. With the efficient implementations of non-linear data structures shown below, you can operate on the data in a quicker fashion, thereby speeding up the algorithms that rely on them.

For example, if you need a *dynamic*[11] collection of pairs (e.g. key → value pairs), using C++ STL `map` below can provide you $O(\log n)$ performance for insertion/search/deletion operations with just a few lines of code (that you still have to write yourself), whereas storing the same information inside a static array of `struct`s may require $O(n)$ insertion/search/deletion, and you will need to write the longer traversal code yourself.

- Balanced Binary Search Tree (BST): C++ STL `map/set` (Java `TreeMap/TreeSet`)
  The BST is one way to organize data in a tree structure. In each subtree rooted at x, the following BST property holds: Items on the left subtree of x are smaller than x and items on the right subtree of x are greater than (or equal to) x. This is essentially an application of the Divide and Conquer strategy (also see Section 3.3). Organizing the data like this (see Figure 2.2) allows for $O(\log n)$ `search(key)`, `insert(key)`, `findMin()/findMax()`, `successor(key)/predecessor(key)`, and `delete(key)` since in the worst case, only $O(\log n)$ operations are required in a root-to-leaf scan (see [7, 5, 54, 12] for details). However, this only holds if the BST is balanced.

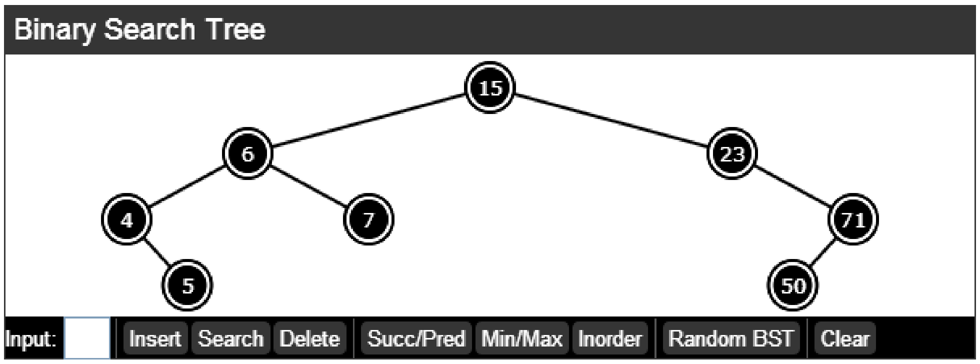


Figure 2.2: Examples of BST

  Implementing *bug-free* balanced BSTs such as the Adelson-Velskii Landis (AVL)[12] or Red-Black (RB)[13] Trees is a tedious task and is difficult to achieve in a time-constrained contest environment (unless you have prepared a code library beforehand, see Section 9.29). Fortunately, C++ STL has `map` and `set` (and Java has `TreeMap` and `TreeSet`) which are *usually* implementations of the RB Tree which guarantees that major BST operations like insertions/searches/deletions are done in $O(\log n)$ time. By mastering these two C++ STL template classes (or Java APIs), you can save a lot of precious coding time during contests! The difference between these two data structures is simple: the C++ STL `map` (and Java `TreeMap`) stores (key → data) pairs whereas the C++

[11]The contents of a dynamic data structure is frequently modified via insert/delete/update operations.

[12]The AVL tree was the first self-balancing BST to be invented. AVL trees are essentially traditional BSTs with an additional property: The heights of the two subtrees of any vertex in an AVL tree can differ *by at most one*. Rebalancing operations (rotations) are performed (when necessary) during insertions and deletions to maintain this invariant property, hence keeping the tree roughly balanced.

[13]The Red-Black tree is another self-balancing BST, in which every vertex has a color: red or black. In RB trees, the root vertex, all leaf vertices, and both children of every red vertex are black. Every simple path from a vertex to any of its descendant leaves contains *the same number of black vertices*. Throughout insertions and deletions, an RB tree will maintain all these invariants to keep the tree balanced.

STL `set` (and Java `TreeSet`) only stores the key. For most (contest) problems, we use a `map` (to really map information) instead of a `set` (a `set` is only useful for efficiently determining the existence of a certain key). However, there is a small drawback. If we use the library implementations, it becomes difficult or impossible to augment (add extra information to) the BST. Please attempt **Exercise 2.3.5*** and read Section 9.29 for more details.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bst.html`

Source code: `ch2_05_map_set.cpp/java`

- Heap: C++ STL `priority_queue` (Java `PriorityQueue`)
The heap is another way to organize data in a tree. The (Binary) Heap is also a binary tree like the BST, except that it must be a *complete*[14] tree. Complete binary trees can be stored efficiently in a compact 1-indexed array of size $n + 1$, which is often preferred to an explicit tree representation. For example, the array A = {N/A, 90, 19, 36, 17, 3, 25, 1, 2, 7} is the compact array representation of Figure 2.3 with index 0 ignored. One can navigate from a certain index (vertex) i to its parent, left child, and right child by using simple index manipulation: $\lfloor \frac{i}{2} \rfloor$, $2 \times i$, and $2 \times i + 1$, respectively. These index manipulations can be made faster using bit manipulation techniques (see Section 2.2): `i >> 1`, `i << 1`, and `(i << 1) + 1`, respectively.

  Instead of enforcing the BST property, the (Max) Heap enforces the Heap property: in each subtree rooted at x, items on the left **and** right subtrees of x are smaller than (or equal to) x (see Figure 2.3). This is also an application of the Divide and Conquer concept (see Section 3.3). The property guarantees that the top (or root) of the heap is always the maximum element. There is no notion of a 'search' in the Heap (unlike BSTs). The Heap instead allows for the fast extraction (deletion) of the maximum element: `ExtractMax()` and insertion of new items: `Insert(v)`—both of which can be easily achieved by in a $O(\log n)$ root-to-leaf or leaf-to-root traversal, performing swapping operations to maintain the (Max) Heap property whenever necessary (see [7, 5, 54, 12] for details).

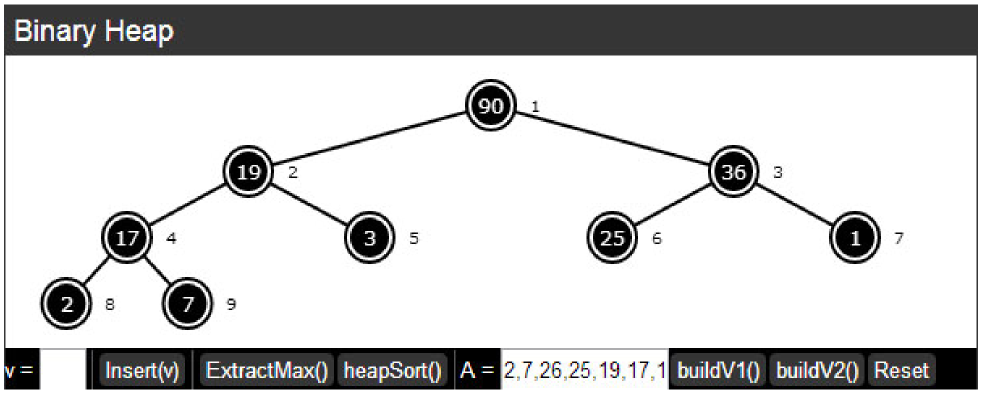


Figure 2.3: (Max) Heap Visualization

  The (Max) Heap is a useful data structure for modeling a Priority Queue, where the item with the highest priority (the maximum element) can be dequeued (`ExtractMax()`)

[14]A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled. All vertices in the last level must also be filled from left-to-right.

and a new item v can be enqueued (`Insert(v)`), both in $O(\log n)$ time. The implementation[15] of `priority_queue` is available in the C++ STL `queue` library (or Java `PriorityQueue`). Priority Queues are an important component in algorithms like Prim's (and Kruskal's) algorithms for the Minimum Spanning Tree (MST) problem (see Section 4.3), Dijkstra's algorithm for the Single-Source Shortest Paths (SSSP) problem (see Section 4.4.3), and the A* Search algorithm (see Section 8.2.5).

This data structure is also used to perform `partial_sort` in the C++ STL `algorithm` library. One possible implementation is by processing the elements one by one and creating a Max[16] Heap of k elements, removing the largest element whenever its size exceeds k (k is the number of elements requested by user). The smallest k elements can then be obtained in descending order by dequeuing the remaining elements in the Max Heap. As each dequeue operation is $O(\log k)$, `partial_sort` has $O(n \log k)$ time complexity[17]. When $k = n$, this algorithm is equivalent to a heap sort. Note that although the time complexity of a heap sort is also $O(n \log n)$, heap sorts are often slower than quick sorts because heap operations access data stored in distant indices and are thus not cache-friendly.

Visualization: `www.comp.nus.edu.sg/∼stevenha/visualization/heap.html`

Source code: `ch2_06_priority_queue.cpp/java`

- Hash Table: C++11 STL `unordered_map`[18] (and Java `HashMap`/`HashSet`/`HashTable`)
  The Hash Table is another non-linear data structure, but we do not recommend using it in programming contests unless absolutely necessary. Designing a well-performing hash function is often tricky and only the new C++11 has STL support for it (Java has Hash-related classes).

  Moreover, C++ STL `map`s or `set`s (and Java `TreeMap`s or `TreeSet`s) are usually fast enough as the typical input size of (programming contest) problems is usually not more than $1M$. Within these bounds, the $O(1)$ performance of Hash Tables and $O(\log 1M)$ performance for balanced BSTs do not differ by much. Thus, we do not discuss Hash Tables in detail in this section.

  However, a simple form of Hash Tables can be used in programming contests. 'Direct Addressing Tables' (DATs) can be considered to be Hash Tables where the keys themselves are the indices, or where the 'hash function' is the identity function. For example, we may need to assign all possible ASCII characters [0-255] to integer values, e.g. 'a' → '3', 'W' → '10', ..., 'I' → '13'. For this purpose, we do not need the C++ STL `map` or any form of hashing as the key itself (the value of the ASCII character) is unique and sufficient to determine the appropriate index in an array of size 256. Some programming exercises involving DATs are listed in the previous Section 2.2.

[15]The default C++ STL `priority_queue` is a Max Heap (dequeuing yields items in descending key order) whereas the default Java `PriorityQueue` is a Min Heap (yields items in ascending key order). Tips: A Max Heap containing numbers can easily be converted into a Min Heap (and vice versa) by inserting the negated keys. This is because negating a set of numbers will reverse their order of appearance when sorted. This trick is used several times in this book. However, if the priority queue is used to store *32-bit signed integers*, an overflow will occur if -2^{31} is negated as $2^{31} - 1$ is the maximum value of a 32-bit signed integer.

[16]The default `partial_sort` produces the smallest k elements in ascending order.

[17]You may have noticed that the time complexity $O(n \log k)$ where k is the output size and n is the input size. This means that the algorithm is 'output-sensitive' since its running time depends not only on the input size but also on the amount of items that it has to output.

[18]Note that C++11 is a new C++ standard, older compilers may not support it yet.

Exercise 2.3.1: Someone suggested that it is possible to store the key → value pairs in a *sorted array* of `structs` so that we can use the $O(\log n)$ binary search for the example problem above. Is this approach feasible? If no, what is the issue?

Exercise 2.3.2: We will not discuss the basics of BST operations in this book. Instead, we will use a series of sub-tasks to verify your understanding of BST-related concepts. We will use Figure 2.2 as an *initial reference* in all sub-tasks except sub-task 2.

1. Display the steps taken by `search(71)`, `search(7)`, and then `search(22)`.
2. Starting with an *empty* BST, display the steps taken by `insert(15)`, `insert(23)`, `insert(6)`, `insert(71)`, `insert(50)`, `insert(4)`, `insert(7)`, and `insert(5)`.
3. Display the steps taken by `findMin()` (and `findMax()`).
4. Indicate the *inorder traversal* of this BST. Is the output sorted?
5. Display the steps taken by `successor(23)`, `successor(7)`, and `successor(71)`.
6. Display the steps taken by `delete(5)` (a leaf), `delete(71)` (an internal node with one child), and then `delete(15)` (an internal node with two children).

Exercise 2.3.3*: Suppose you are given a reference to the root R of a binary tree T containing n vertices. You can access a node's left, right and parent vertices as well as its key through its reference. Solve each of the following tasks below with the best possible algorithms that you can think of and analyze their time complexities. Let's assume the following constraints: $1 \leq n \leq 100K$ so that $O(n^2)$ solutions are theoretically infeasible in a contest environment.

1. Check if T is a BST.

2*. Output the elements in T that are within a given range $[a..b]$ in ascending order.

3*. Output the contents of the *leaves* of T in *descending order*.

Exercise 2.3.4*: The inorder traversal (also see Section 4.7.2) of a standard (not necessarily balanced) BST is known to produce the BST's element in sorted order and runs in $O(n)$. Does the code below also produce the BST elements in sorted order? Can it be made to run in a total time of $O(n)$ instead of $O(\log n + (n-1) \times \log n) = O(n \log n)$? If possible, how?

```
x = findMin(); output x
for (i = 1; i < n; i++)                        // is this loop O(n log n)?
  x = successor(x); output x
```

Exercise 2.3.5*: Some (hard) problems require us to write *our own* balanced Binary Search Tree (BST) implementations due to the need to augment the BST with additional data (see Chapter 14 of [7]). Challenge: Solve UVa 11849 - CD which is a pure balanced BST problem with *your own* balanced BST implementation to test its performance and correctness.

Exercise 2.3.6: We will not discuss the basics of Heap operations in this book. Instead, we will use a series of questions to verify your understanding of Heap concepts.

1. With Figure 2.3 as the initial heap, display the steps taken by `Insert(26)`.
2. After answering question 1 above, display the steps taken by `ExtractMax()`.

Exercise 2.3.7: Is the structure represented by a 1-based compact array (ignoring index 0) sorted in descending order a Max Heap?

Exercise 2.3.8*: Prove or disprove this statement: "The second largest element in a Max Heap with $n \geq 3$ distinct elements is always one of the direct children of the root". Follow up question: What about the third largest element? Where is/are the potential location(s) of the third largest element in a Max Heap?

Exercise 2.3.9*: Given a 1-based compact array A containing n integers ($1 \leq n \leq 100K$) that are guaranteed to satisfy the Max Heap property, output the elements in A that are greater than an integer v. What is the best algorithm?

Exercise 2.3.10*: Given an unsorted array S of n distinct integers ($2k \leq n \leq 100000$), find the largest and smallest k ($1 \leq k \leq 32$) integers in S in $O(n \log k)$. Note: For this written exercise, assume that an $O(n \log n)$ algorithm is *not* acceptable.

Exercise 2.3.11*: One heap operation *not* directly supported by the C++ STL `priority_queue` (and Java `PriorityQueue`) is the `UpdateKey(index, newKey)` operation, which allows the (Max) Heap element at a certain index to be updated (increased or decreased). Write *your own* binary (Max) Heap implementation with this operation.

Exercise 2.3.12*: Another heap operation that may be useful is the `DeleteKey(index)` operation to delete (Max) Heap elements at a certain index. Implement this!

Exercise 2.3.13*: Suppose that we only need the `DecreaseKey(index, newKey)` operation, i.e. an `UpdateKey` operation where the update *always* makes `newKey` smaller than its previous value. Can we use a simpler approach than in **Exercise 2.3.11**? Hint: Use lazy deletion, we will use this technique in our Dijkstra code in Section 4.4.3.

Exercise 2.3.14*: Is it possible to use a balanced BST (e.g. C++ STL `set` or Java `TreeSet`) to implement a Priority Queue with the same $O(\log n)$ enqueue and dequeue performance? If yes, how? Are there any potential drawbacks? If no, why?

Exercise 2.3.15*: Is there a better way to implement a Priority Queue if the keys are all integers within a small range, e.g. $[0 \ldots 100]$? We are expecting an $O(1)$ enqueue and dequeue performance. If yes, how? If no, why?

Exercise 2.3.16: Which non-linear data structure should you use if you have to support the following three dynamic operations: 1) many insertions, 2) many deletions, and 3) many requests for the data in sorted order?

Exercise 2.3.17: There are M **strings**. N of them are unique ($N \leq M$). Which non-linear data structure discussed in this section should you use if you have to index (label) these **M** strings with integers from `[0..N-1]`? The indexing criteria is as follows: The first string must be given an index of 0; The next different string must be given index 1, and so on. However, if a string is re-encountered, it must be given the same index as its earlier copy! One application of this task is in constructing the connection graph from a list of city names (which are not integer indices!) and a list of highways between these cities (see Section 2.4.1). To do this, we first have to map these city names into integer indices (which are far more efficient to work with).

Programming exercises solvable with library of non-linear data structures:

- C++ STL `map` (and Java `TreeMap`)
    1. UVa 00417 - Word Index (generate all words, add to `map` for auto sorting)
    2. UVa 00484 - The Department of ... (maintain frequency with `map`)
    3. UVa 00860 - Entropy Text Analyzer (frequency counting)
    4. *UVa 00939 - Genes* (`map` child name to his/her gene and parents' names)
    5. *UVa 10132 - File Fragmentation* (N = number of fragments, B = total bits of all fragments divided by $N/2$; try all $2 \times N^2$ concatenations of two fragments that have length B; report the one with highest frequency; use `map`)
    6. *UVa 10138 - CDVII* (map plates to bills, entrace time and position)
    7. **UVa 10226 - Hardwood Species *** (use hashing for a better performance)
    8. UVa 10282 - Babelfish (a pure dictionary problem; use `map`)
    9. UVa 10295 - Hay Points (use `map` to deal with Hay Points dictionary)
    10. UVa 10686 - SQF Problem (use `map` to manage the data)
    11. UVa 11239 - Open Source (use `map` and `set` to check previous strings)
    12. **UVa 11286 - Conformity *** (use `map` to keep track of the frequencies)
    13. UVa 11308 - Bankrupt Baker (use `map` and `set` to help manage the data)
    14. *UVa 11348 - Exhibition* (use `map` and `set` to check uniqueness)
    15. ***UVa 11572 - Unique Snowflakes **** (use `map` to record the occurrence index of a certain snowflake size; use this to determine the answer in $O(n \log n)$)
    16. UVa 11629 - Ballot evaluation (use `map`)
    17. UVa 11860 - Document Analyzer (use `set` and `map`, linear scan)
    18. UVa 11917 - Do Your Own Homework (use `map`)
    19. *UVa 12504 - Updating a Dictionary* (use `map`; string to string; a bit tedious)
    20. *UVa 12592 - Slogan Learning of Princess* (use `map`; string to string)

    Also check frequency counting section in Section 6.3.
- C++ STL `set` (Java `TreeSet`)
    1. UVa 00501 - Black Box (use `multiset` with efficient iterator manipulation)
    2. **UVa 00978 - Lemmings Battle *** (simulation, use `multiset`)
    3. UVa 10815 - Andy's First Dictionary (use `set` and `string`)
    4. UVa 11062 - Andy's Second Dictionary (similar to UVa 10815, with twists)
    5. **UVa 11136 - Hoax or what *** (use `multiset`)
    6. **UVa 11849 - CD *** (use `set` to pass the time limit, better: use hashing!)
    7. *UVa 12049 - Just Prune The List* (`multiset` manipulation)
- C++ STL `priority_queue` (Java `PriorityQueue`)
    1. **UVa 01203 - Argus *** (LA 3135, Beijing04; use `priority_queue`)
    2. **UVa 10954 - Add All *** (use `priority_queue`, greedy)
    3. **UVa 11995 - I Can Guess ... *** (`stack`, `queue`, and `priority_queue`)

    Also see the usage of `priority_queue` for topological sorts (see Section 4.2.1), Kruskal's[19] (see Section 4.3.2), Prim's (see Section 4.3.3), Dijkstra's (see Section 4.4.3), and the A* Search algorithms (see Section 8.2.5)

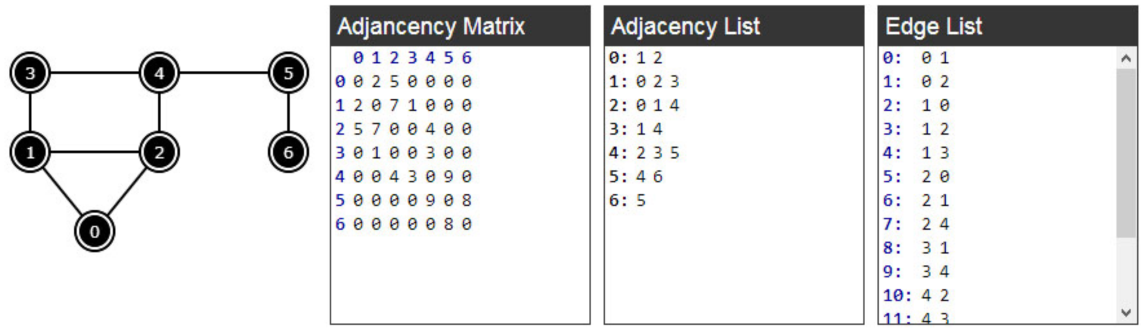
[19]This is another way to implement the edge sorting in Kruskal's algorithm. Our (C++) implementation shown in Section 4.3.2 simply uses `vector` + `sort` instead of `priority_queue` (a heap sort).

2.4 Data Structures with Our Own Libraries

As of 24 May 2013, important data structures shown in this section do not have built-in support yet in C++ STL or Java API. Thus, to be competitive, contestants should prepare bug-free implementations of these data structures. In this section, we discuss the key ideas and example implementations (see the given source code too) of these data structures.

2.4.1 Graph

The graph is a pervasive structure which appears in many Computer Science problems. A graph ($G = (V, E)$) in its basic form is simply a set of vertices (V) and edges (E; storing connectivity information between vertices in V). Later in Chapter 3, 4, 8, and 9, we will explore many important graph problems and algorithms. To prepare ourselves, we will discuss three basic ways (there are a few other rare structures) to represent a graph G with V vertices and E edges in this subsection[20].



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

```
0: 1 2
1: 0 2 3
2: 0 1 4
3: 1 4
4: 2 3 5
5: 4 6
6: 5
```

```
0:  0 1
1:  0 2
2:  1 0
3:  1 2
4:  1 3
5:  2 0
6:  2 1
7:  2 4
8:  3 1
9:  3 4
10: 4 2
11: 4 3
```

Figure 2.4: Graph Data Structure Visualization

A). The Adjacency Matrix, usually in the form of a 2D array (see Figure 2.4).

In (programming contest) problems involving graphs, the number of vertices V is usually known. Thus we can build a 'connectivity table' by creating a static 2D array: `int AdjMat[`V`][`V`]`. This has an $O(V^2)$ *space*[21] complexity. For an unweighted graph, set `AdjMat[i][j]` to a non-zero value (usually 1) if there is an edge between vertex i-j or zero otherwise. For a weighted graph, set `AdjMat[i][j] = weight(i,j)` if there is an edge between vertex i-j with `weight(i,j)` or zero otherwise. Adjacency Matrix cannot be used to store multigraph. For a simple graph without self-loops, the main diagonal of the matrix contains only zeroes, i.e. `AdjMat[i][i] = 0,` $\forall$`i` $\in$ `[0..V-1]`.

An Adjacency Matrix is a good choice if the connectivity between two vertices in a *small dense graph* is frequently required. However, it is not recommended for *large sparse graphs* as it would require too much space ($O(V^2)$) and there would be many blank (zero) cells in the 2D array. In a competitive setting, it is usually infeasible to use Adjacency Matrices when the given V is larger than $\approx$ 1000. Another drawback of Adjacency Matrix is that it also takes $O(V)$ time to enumerate the list of neighbors of a vertex v—an operation common to many graph algorithms—even if a vertex only has a handful of neighbors. A more compact and efficient graph representation is the Adjacency List discussed below.

[20]The most appropriate notation for the cardinality of a set S is $|S|$. However, in this book, we will often overload the meaning of V or E to also mean $|V|$ or $|E|$, depending on the context.

[21]We differentiate between the *space* and *time* complexities of data structures. The *space* complexity is an asymptotic measure of the memory requirements of a data structure whereas the *time* complexity is an asymptotic measure of the time taken to run a certain algorithm or an operation on the data structure.

B). The Adjacency List, usually in the form of a vector of vector of pairs (see Figure 2.4).
Using the C++ STL: `vector<vii> AdjList`, with `vii` defined as in:
`typedef pair<int, int> ii; typedef vector<ii> vii; // data type shortcuts`
Using the Java API: `Vector< Vector < IntegerPair > > AdjList`.
`IntegerPair` is a simple Java class that contains a pair of integers like `ii` above.

In Adjacency Lists, we have a `vector` of `vector` of pairs, storing the list of neighbors of each vertex u as 'edge information' pairs. Each pair contains two pieces of information, the index of the neighbouring vertex and the weight of the edge. If the graph is unweighted, simply store the weight as 0, 1, or drop the weight attribute[22] entirely. The space complexity of Adjacency List is $O(V + E)$ because if there are E bidirectional edges in a (simple) graph, this Adjacency List will only store $2E$ 'edge information' pairs. As E is usually much smaller than $V \times (V - 1)/2 = O(V^2)$—the maximum number of edges in a complete (simple) graph, Adjacency Lists are often more space-efficient than Adjacency Matrices. Note that Adjacency List can be used to store multigraph.

With Adjacency Lists, we can also enumerate the list of neighbors of a vertex v efficiently. If v has k neighbors, the enumeration will require $O(k)$ time. Since this is one of the most common operations in most graph algorithms, it is advisable to use Adjacency Lists as your first choice of graph representation. Unless otherwise stated, most graph algorithms discussed in this book use the Adjacency List.

C). The Edge List, usually in the form of a vector of triples (see Figure 2.4).
Using the C++ STL: `vector< pair<int, ii> > EdgeList`.
Using the Java API: `Vector< IntegerTriple > EdgeList`.
`IntegerTriple` is a class that contains a triple of integers like `pair<int, ii>` above.

In the Edge List, we store a list of all E edges, usually in some sorted order. For directed graphs, we can store a bidirectional edge twice, one for each direction. The space complexity is clearly $O(E)$. This graph representation is very useful for Kruskal's algorithm for MST (Section 4.3.2), where the collection of undirected edges need to be sorted[23] by ascending weight. However, storing graph information in Edge List complicates many graph algorithms that require the enumeration of edges incident to a vertex.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/graphds.html`

Source code: `ch2_07_graph_ds.cpp/java`

Implicit Graph

Some graphs do *not* have to be stored in a graph data structure or explicitly generated for the graph to be traversed or operated upon. Such graphs are called *implicit* graphs. You will encounter them in the subsequent chapters. Implicit graphs can come in two flavours:

1. The edges can be determined easily.

   Example 1: Navigating a 2D grid map (see Figure 2.5.A). The vertices are the cells in the 2D character grid where '.' represents land and '#' represents an obstacle. The edges can be determined easily: There is an edge between two neighboring cells in the

[22]For simplicity, we will always assume that the second attribute exists in all graph implementations in this book although it is not always used.

[23]`pair` objects in C++ can be easily sorted. The default sorting criteria is to sort on the first item and then the second item for tie-breaking. In Java, we can write our own `IntegerPair/IntegerTriple` class that implements `Comparable`.

grid if they share an N/S/E/W border and if both are '.' (see Figure 2.5.B).

Example 2: The graph of chess knight movements on an 8×8 chessboard. The vertices are the cells in the chessboard. Two squares in the chessboard have an edge between them if they differ by two squares horizontally and one square vertically (or two squares vertically and one square horizontally). The first three rows and four columns of a chessboard are shown in Figure 2.5.C (many other vertices and edges are not shown).

2. The edges can be determined with some rules.

   Example: A graph contains N vertices (`[1..N]`). There is an edge between two vertices i and j if $(i + j)$ is a prime. See Figure 2.5.D that shows such a graph with $N = 5$ and several more examples in Section 8.2.3.

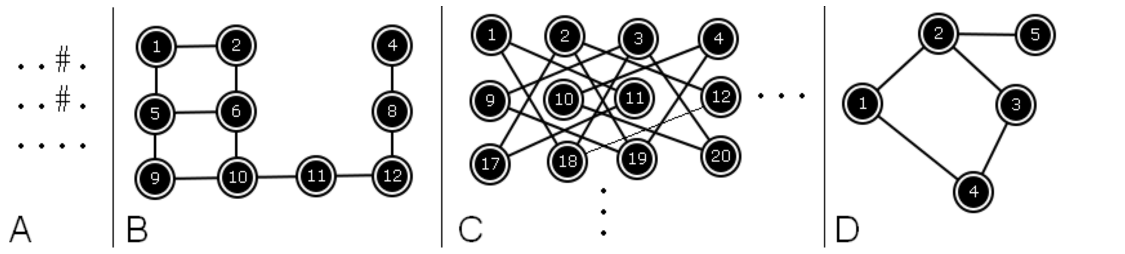


Figure 2.5: Implicit Graph Examples

Exercise 2.4.1.1*: Create the Adjacency Matrix, Adjacency List, and Edge List representations of the graphs shown in Figure 4.1 (Section 4.2.1) and in Figure 4.9 (Section 4.2.9). Hint: Use the graph data structure visualization tool shown above.

Exercise 2.4.1.2*: Given a (simple) graph in one representation (Adjacency Matrix/AM, Adjacency List/AL, or Edge List/EL), *convert* it into another graph representation in the most efficient way possible! There are 6 possible conversions here: AM to AL, AM to EL, AL to AM, AL to EL, EL to AM, and EL to AL.

Exercise 2.4.1.3: If the Adjacency Matrix of a (simple) graph has the property that it is equal to its transpose, what does this imply?

Exercise 2.4.1.4*: Given a (simple) graph represented by an Adjacency Matrix, perform the following tasks in the most efficient manner. Once you have figured out how to do this for Adjacency Matrices, perform the same task with Adjacency Lists and then Edge Lists.

1. Count the number of vertices V and directed edges E (assume that a bidirectional edge is equivalent to two directed edges) of the graph.

2*. Count the in-degree and the out-degree of a certain vertex v.

3*. Transpose the graph (reverse the direction of each edges).

4*. Check if the graph is a complete graph K_n. Note: A complete graph is a simple undirected graph in which *every pair* of distinct vertices is connected by a single edge.

5*. Check if the graph is a tree (a connected undirected graph with $E = V - 1$ edges).

6*. Check if the graph is a star graph S_k. Note: A star graph S_k is a complete bipartite $K_{1,k}$ graph. It is a tree with only one internal vertex and k leaves.

Exercise 2.4.1.5*: Research other possible methods of representing graphs other than the ones discussed above, especially for storing special graphs!

2.4.2 Union-Find Disjoint Sets

The Union-Find Disjoint Set (UFDS) is a data structure to model a collection of *disjoint sets* with the ability to efficiently[24]—in $\approx O(1)$—determine which set an item belongs to (or to test whether two items belong to the same set) and to unite two disjoint sets into one larger set. Such data structure can be used to solve the problem of finding connected components in an undirected graph (Section 4.2.3). Initialize each vertex to a separate disjoint set, then enumerate the graph's edges and join every two vertices/disjoint sets connected by an edge. We can then test if two vertices belong to the same component/set easily.

These seemingly simple operations are not *efficiently* supported by the C++ STL `set` (and Java `TreeSet`), which is not designed for this purpose. Having a `vector` of `set`s and looping through each one to find which set an item belongs to is expensive! C++ STL `set_union` (in `algorithm`) will not be efficient enough although it combines two sets in *linear time* as we still have to deal with shuffling the contents of the `vector` of `set`s! To support these set operations efficiently, we need a better data structure—the UFDS.

The main innovation of this data structure is in choosing a representative 'parent' item to represent a set. If we can ensure that each set is represented by only one unique item, then determining if items belong to the same set becomes far simpler: The representative 'parent' item can be used as a sort of identifier for the set. To achieve this, the Union-Find Disjoint Set creates a tree structure where the disjoint sets form a forest of trees. Each tree corresponds to a disjoint set. The root of the tree is determined to be the representative item for a set. Thus, the representative set identifier for an item can be obtained simply by following the chain of parents to the root of the tree, and since a tree can only have one root, this representative item can be used as a unique identifier for the set.

To do this efficiently, we store the index of the parent item and (the upper bound of) the height of the tree of each set (`vi p` and `vi rank` in our implementation). Remember that `vi` is our shortcut for a vector of integers. `p[i]` stores the immediate parent of item `i`. If item `i` is the representative item of a certain disjoint set, then `p[i] = i`, i.e. a self-loop. `rank[i]` yields (the upper bound of) the height of the tree rooted at item `i`.

In this section, we will use 5 disjoint sets {0, 1, 2, 3, 4} to illustrate the usage of this data structure. We initialize the data structure such that each item is a disjoint set by itself with rank 0 and the parent of each item is initially set to itself.

To unite two disjoint sets, we set the representative item (root) of one disjoint set to be the new parent of the representative item of the other disjoint set. This effectively merges the two trees in the Union-Find Disjoint Set representation. As such, `unionSet(i, j)` will cause both items '`i`' and '`j`' to have the same representative item—directly or indirectly. For efficiency, we can use the information contained in `vi rank` to set the representative item of the disjoint set with *higher rank* to be the new parent of the disjoint set with *lower rank*, thereby `minimizing` the rank of the resulting tree. If both ranks are the same, we arbitrarily choose one of them as the new parent and increase the resultant root's rank. This is the 'union by rank' heuristic. In Figure 2.6, top, `unionSet(0, 1)` sets `p[0]` to 1 and `rank[1]` to 1. In Figure 2.6, middle, `unionSet(2, 3)` sets `p[2]` to 3 and `rank[3]` to 1.

For now, let's assume that function `findSet(i)` simply calls `findSet(p[i])` recursively to find the representative item of a set, returning `findSet(p[i])` whenever `p[i] != i` and `i` otherwise. In Figure 2.6, bottom, when we call `unionSet(4, 3)`, we have `rank[findSet(4)] = rank[4] = 0` which is smaller than `rank[findSet(3)] = rank[3] = 1`, so we set `p[4] = 3` *without* changing the height of the resulting tree—this is the 'union by rank' heuristic

[24] M operations of this UFDS data structure with 'path compression' and 'union by rank' heuristics run in $O(M \times \alpha(n))$. However, since the inverse Ackermann function $\alpha(n)$ grows very slowly, i.e. its value is just less than 5 for practical input size $n \leq 1M$ in programming contest setting, we can treat $\alpha(n)$ as constant.

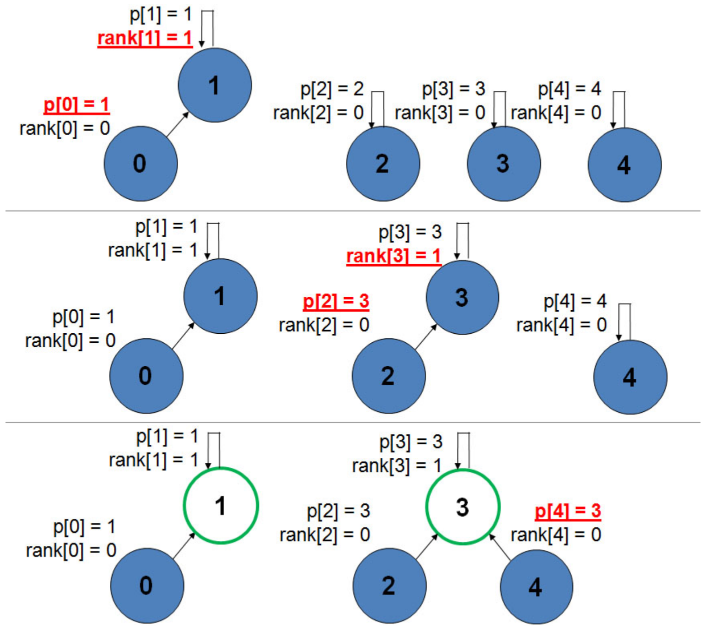


Figure 2.6: `unionSet(0, 1)` → `(2, 3)` → `(4, 3)` and `isSameSet(0, 4)`

at work. With the heuristic, the path taken from any node to the representative item by following the chain of 'parent' links is effectively minimized.

In Figure 2.6, bottom, `isSameSet(0, 4)` demonstrates another operation for this data structure. This function `isSameSet(i, j)` simply calls `findSet(i)` and `findSet(j)` and checks if both refer to the same representative item. If they do, then '`i`' and '`j`' both belong to the same set. Here, we see that `findSet(0) = findSet(p[0]) = findSet(1) = 1` is not the same as `findSet(4)= findSet(p[4]) = findSet(3) = 3`. Therefore item 0 and item 4 belongs to *different* disjoint sets.

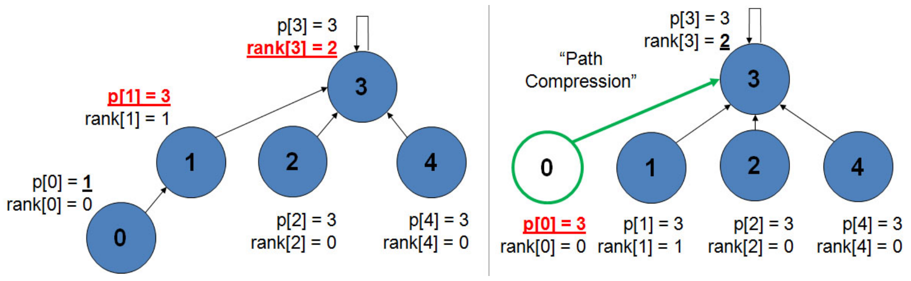


Figure 2.7: `unionSet(0, 3)` → `findSet(0)`

There is a technique that can vastly speed up the `findSet(i)` function: Path compression. Whenever we find the representative (root) item of a disjoint set by following the chain of 'parent' links from a given item, we can set the parent of *all items* traversed to point directly to the root. Any subsequent calls to `findSet(i)` on the affected items will then result in only one link being traversed. This changes the structure of the tree (to make `findSet(i)` more efficient) but yet preserves the actual constitution of the disjoint set. Since this will occur any time `findSet(i)` is called, the combined effect is to render the runtime of the `findSet(i)` operation to run in an extremely efficient amortized $O(M \times \alpha(n))$ time.

In Figure 2.7, we demonstrate this 'path compression'. First, we call `unionSet(0, 3)`. This time, we set `p[1] = 3` and update `rank[3] = 2`. Now notice that `p[0]` is unchanged, i.e. `p[0] = 1`. This is an *indirect* reference to the (true) representative item of the set, i.e. `p[0] = 1` → `p[1] = 3`. Function `findSet(i)` will actually require more than one step to traverse the chain of 'parent' links to the root. However, once it finds the representative item, (e.g. 'x') for that set, it will *compress the path* by setting `p[i] = x`, i.e. `findSet(0)` sets `p[0] = 3`. Therefore, subsequent calls of `findSet(i)` will be just $O(1)$. This simple strategy is aptly named the 'path compression' heuristic. Note that `rank[3] = 2` now no longer reflects the *true height* of the tree. This is why `rank` only reflects the *upper bound* of the actual height of the tree. Our C++ implementation is shown below:

```
class UnionFind {                                                // OOP style
private: vi p, rank;                              // remember: vi is vector<int>
public:
  UnionFind(int N) { rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) {                          // if from different set
      int x = findSet(i), y = findSet(j);
      if (rank[x] > rank[y]) p[y] = x;          // rank keeps the tree short
      else {                 p[x] = y;
                             if (rank[x] == rank[y]) rank[y]++; }
} } };
```

Visualization: `www.comp.nus.edu.sg/∼stevenha/visualization/ufds.html`

Source code: `ch2_08_unionfind_ds.cpp/java`

Exercise 2.4.2.1: There are two more queries that are commonly performed in this data structure. Update the code provided in this section to support these two queries efficiently: `int numDisjointSets()` that returns the number of disjoint sets currently in the structure and `int sizeOfSet(int i)` that returns the size of set that currently contains item `i`.

Exercise 2.4.2.2*: Given 8 disjoint sets: {0, 1, 2, ..., 7}, please create a sequence of `unionSet(i, j)` operations to create a tree with rank = 3! Is this possible for rank = 4?

Profiles of Data Structure Inventors

George Boole (1815-1864) was an English mathematician, philosopher, and logician. He is best known to Computer Scientists as the founder of Boolean logic, the foundation of modern digital computers. Boole is regarded as the founder of the field of Computer Science.

Rudolf Bayer (born 1939) has been Professor (emeritus) of Informatics at the Technical University of Munich. He invented the Red-Black (RB) tree used in the C++ STL `map/set`.

Georgii Adelson-Velskii (born 1922) is a Soviet mathematician and computer scientist. Along with Evgenii Mikhailovich Landis, he invented the AVL tree in 1962.

Evgenii Mikhailovich Landis (1921-1997) was a Soviet mathematician. The name of the AVL tree is an abbreviation of the two inventors: Adelson-Velskii and Landis himself.

2.4.3 Segment Tree

In this subsection, we will discuss a data structure which can efficiently answer *dynamic*[25] range queries. One such range query is the problem of finding the index of the minimum element in an array within range `[i..j]`. This is more commonly known as the Range Minimum Query (RMQ) problem. For example, given an array `A` of size $n = 7$ below, `RMQ(1, 3) = 2`, as the index 2 contains the minimum element among `A[1]`, `A[2]`, and `A[3]`. To check your understanding of RMQ, verify that in the array A below, `RMQ(3, 4) = 4`, `RMQ(0, 0) = 0`, `RMQ(0, 1) = 1`, and `RMQ(0, 6) = 5`. For the next few paragraphs, assume that array A is the same.

Array	Values	18	17	13	19	15	11	20
A	Indices	0	1	2	3	4	5	6

There are several ways to implement the RMQ. One trivial algorithm is to simply iterate the array from index `i` to `j` and report the index with the minimum value, but this will run in $O(n)$ time per query. When n is large and there are many queries, such an algorithm may be infeasible.

In this section, we answer the dynamic RMQ problem with a Segment Tree, which is another way to arrange data in a binary tree. There are several ways to implement the Segment Tree. Our implementation uses the same concept as the 1-based compact array in the binary heap where we use `vi` (our shortcut for `vector<int>`) `st` to represent the binary tree. Index 1 (skipping index 0) is the root and the left and right children of index p are index $2 \times p$ and $(2 \times p) + 1$ respectively (also see Binary Heap discussion in Section 2.3). The value of `st[p]` is the RMQ value of the segment associated with index p.

The root of segment tree represents segment `[0, n-1]`. For each segment `[L, R]` stored in index p where `L != R`, the segment will be split into `[L, (L+R)/2]` and `[(L+R)/2+1, R]` in a left and right vertices. The left sub-segment and right sub-segment will be stored in index $2{\times}p$ and $(2{\times}p){+}1$ respectively. When `L = R`, it is clear that `st[p] = L` (or `R`). Otherwise, we will recursively build the segment tree, comparing the minimum value of the left and the right sub-segments and updating the `st[p]` of the segment. This process is implemented in the `build` routine below. This `build` routine creates up to $O(1+2+4+8+\ldots+2^{\log_2 n}) = O(2n)$ (smaller) segments and therefore runs in $O(n)$. However, as we use simple 1-based compact array indexing, we need `st` to be at least of size $2 * 2^{\lfloor(\log_2(n)\rfloor+1}$. In our implementation, we simply use a loose upper bound of space complexity $O(4n) = O(n)$. For array `A` above, the corresponding segment tree is shown in Figure 2.8 and 2.9.

With the segment tree ready, answering an RMQ can be done in $O(\log n)$. The answer for `RMQ(i, i)` is trivial—simply return `i` itself. However, for the general case `RMQ(i, j)`, further checks are needed. Let `p1 = RMQ(i, (i+j)/2)` and `p2 = RMQ((i+j)/2 + 1, j)`. Then `RMQ(i, j)` is `p1` if `A[p1]` $\leq$ `A[p2]` or `p2` otherwise. This process is implemented in the `rmq` routine below.

Take for example the query `RMQ(1, 3)`. The process in Figure 2.8 is as follows: Start from the root (index 1) which represents segment `[0, 6]`. We cannot use the stored minimum value of segment `[0, 6] = st[1] = 5` as the answer for `RMQ(1, 3)` since it is the minimum value over a larger[26] segment than the desired `[1, 3]`. From the root, we only have to go to the left subtree as the root of the right subtree represents segment `[4, 6]` which is outside[27] the desired range in `RMQ(1, 3)`.

[25]For dynamic problems, we need to frequently *update* and query the data. This makes pre-processing techniques useless.

[26]Segment `[L, R]` is said to be larger than query range `[i, j]` if `[L, R]` is not outside the query range and not inside query range (see the other footnotes).

[27]Segment `[L, R]` is said to be outside query range `[i, j]` if `i` $>$ `R || j` $<$ `L`.

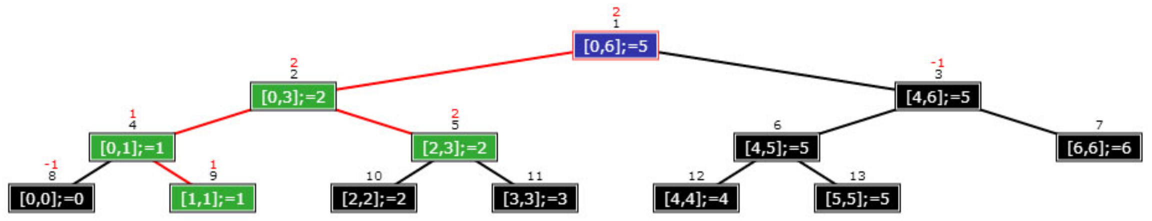


Figure 2.8: Segment Tree of Array A = {18, 17, 13, 19, 15, 11, 20} and `RMQ(1, 3)`

We are now at the root of the left subtree (index 2) that represents segment `[0, 3]`. This segment `[0, 3]` is still larger than the desired `RMQ(1, 3)`. In fact, `RMQ(1, 3)` intersects *both* the left sub-segment `[0, 1]` (index 4) and the right sub-segment `[2, 3]` (index 5) of segment `[0, 3]`, so we have to explore *both* subtrees (sub-segments).

The left segment `[0, 1]` (index 4) of `[0, 3]` (index 2) is not yet inside the `RMQ(1, 3)`, so another split is necessary. From segment `[0, 1]` (index 4), we move right to segment `[1, 1]` (index 9), which is now inside[28] `[1, 3]`. At this point, we know that `RMQ(1, 1) = st[9] = 1` and we can return this value to the caller. The right segment `[2, 3]` (index 5) of `[0, 3]` (index 2) is inside the required `[1, 3]`. From the stored value inside this vertex, we know that `RMQ(2, 3) = st[5] = 2`. We do *not* need to traverse further down.

Now, back in the call to segment `[0, 3]` (index 2), we now have `p1 = RMQ(1, 1) = 1` and `p2 = RMQ(2, 3) = 2`. Because `A[p1]` $>$ `A[p2]` since `A[1] = 17` and `A[2] = 13`, we now have `RMQ(1, 3) = p2 = 2`. This is the final answer.

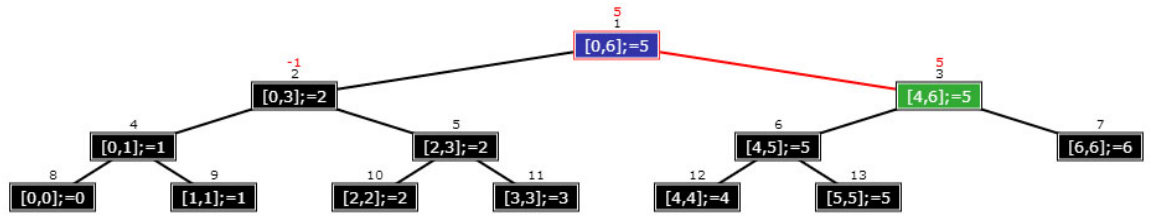


Figure 2.9: Segment Tree of Array A = {18, 17, 13, 19, 15, 11, 20} and `RMQ(4, 6)`

Now let's take a look at another example: `RMQ(4, 6)`. The execution in Figure 2.9 is as follows: We again start from the root segment `[0, 6]` (index 1). Since it is larger than the `RMQ(4, 6)`, we move right to segment `[4, 6]` (index 3) as segment `[0, 3]` (index 2) is outside. Since this segment exactly represents `RMQ(4, 6)`, we simply return the index of minimum element that is stored in this vertex, which is 5. Thus `RMQ(4, 6) = st[3] = 5`.

This data structure allows us to avoid traversing the unnecessary parts of the tree! In the worst case, we have *two* root-to-leaf paths which is just $O(2 \times \log(2n)) = O(\log n)$. Example: In `RMQ(3, 4) = 4`, we have one root-to-leaf path from `[0, 6]` to `[3, 3]` (index $1 \to 2 \to 5 \to 11$) and another root-to-leaf path from `[0, 6]` to `[4, 4]` (index $1 \to 3 \to 6 \to 12$).

If the array `A` is static (i.e. unchanged after it is instantiated), then using a Segment Tree to solve the RMQ problem is *overkill* as there exists a Dynamic Programming (DP) solution that requires $O(n \log n)$ one-time pre-processing and allows for $O(1)$ per RMQ. This DP solution will be discussed later in Section 9.33.

Segment Tree is useful if the underlying array is frequently updated (dynamic). For example, if `A[5]` is now changed from 11 to 99, then we just need to update the vertices along the leaf to root path in $O(\log n)$. See path: `[5, 5]` (index 13, `st[13]` is unchanged) $\to$ `[4, 5]` (index 6, `st[6] = 4` now) $\to$ `[4, 6]` (index 3, `st[3] = 4` now) $\to$ `[0, 6]` (index

[28]Segment `[L, R]` is said to be inside query range `[i, j]` if `L` $\geq$ `i && R` $\leq$ `j`.

1, st[1] = 2 now) in Figure 2.10. For comparison, the DP solution presented in Section 9.33 requires another $O(n \log n)$ pre-processing to update the structure and is ineffective for such dynamic updates.

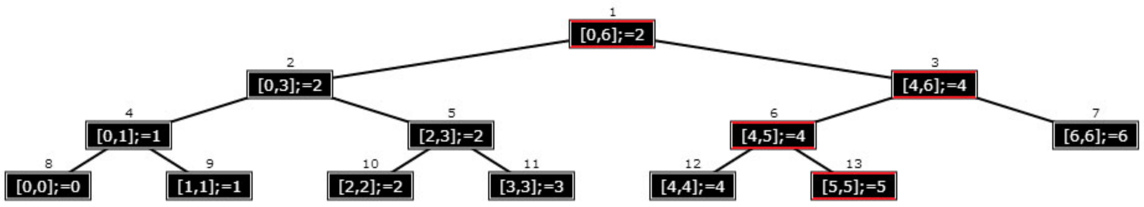


Figure 2.10: Updating Array A to {18, 17, 13, 19, 15, **99**, 20}

Our Segment Tree implementation is shown below. The code shown here supports only *static* RMQs (*dynamic* updates are left as an exercise to the reader).

```
class SegmentTree {         // the segment tree is stored like a heap array
private: vi st, A;            // recall that vi is: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }     // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                                 // O(n)
    if (L == R)                            // as L == R, either one is fine
      st[p] = L;                                         // store the index
    else {                                // recursively compute the values
      build(left(p) , L              , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R          );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }

  int rmq(int p, int L, int R, int i, int j) {                  // O(log n)
    if (i >  R || j <  L) return -1; // current segment outside query range
    if (L >= i && R <= j) return st[p];               // inside query range

     // compute the min position in the left and right part of the interval
    int p1 = rmq(left(p) , L              , (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R          , i, j);

    if (p1 == -1) return p2;   // if we try to access segment outside query
    if (p2 == -1) return p1;                               // same as above
    return (A[p1] <= A[p2]) ? p1 : p2;               // as in build routine
  }

public:
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();              // copy content for local usage
    st.assign(4 * n, 0);            // create large enough vector of zeroes
    build(1, 0, n - 1);                                  // recursive build
  }
  int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }   // overloading
};

int main() {
  int arr[] = { 18, 17, 13, 19, 15, 11, 20 };         // the original array
  vi A(arr, arr + 7);
  SegmentTree st(A);
  printf("RMQ(1, 3) = %d\n", st.rmq(1, 3));            // answer = index 2
  printf("RMQ(4, 6) = %d\n", st.rmq(4, 6));            // answer = index 5
} // return 0;
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/segmenttree.html`

Source code: `ch2_09_segmenttree_ds.cpp/java`

Exercise 2.4.3.1*: Draw the Segment Tree corresponding to array `A = {10, 2, 47, 3, 7, 9, 1, 98, 21}`. Answer RMQ(1, 7) and RMQ(3, 8)! Hint: Use the Segment Tree visualization tool shown above.

Exercise 2.4.3.2*: In this section, we have seen how Segment Trees can be used to answer Range Minimum Queries (RMQs). Segment Trees can also be used to answer dynamic Range Sum Queries (`RSQ(i, j)`), i.e. a sum from `A[i] + A[i + 1] + ...+ A[j]`. Modify the given Segment Tree code above to deal with RSQ.

Exercise 2.4.3.3: Using a similar Segment Tree to **Exercise 2.4.3.1** above, answer the queries `RSQ(1, 7)` and `RSQ(3, 8)`. Is this a good approach to solve the problem if array `A` is never changed? (also see Section 3.5.2).

Exercise 2.4.3.4*: The Segment Tree code shown above lacks the (point) `update` operation as discussed in the body text. Add the $O(\log n)$ `update` function to update the value of a certain index (point) in array `A` and simultaneously update the corresponding Segment Tree!

Exercise 2.4.3.5*: The (point) update operation shown in the body text only changes the value of a certain index in array `A`. What if we delete existing elements of array `A` or insert a new elements into array `A`? Can you explain what will happen with the given Segment Tree code and what you should do to address it?

Exercise 2.4.3.6*: There is also one more important Segment Tree operation that has not yet been discussed, the *range* update operation. Suppose a certain subarray of `A` is updated to a certain common value. Can we update the Segment Tree efficiently? Study and solve UVa 11402 - Ahoy Pirates—a problem that requires range updates.

2.4.4 Binary Indexed (Fenwick) Tree

Fenwick Tree—also known as **Binary Indexed Tree** (BIT)—were invented by *Peter M. Fenwick* in 1994 [18]. In this book, we will use the term Fenwick Tree as opposed to BIT in order to differentiate with the standard *bit manipulations*. The Fenwick Tree is a useful data structure for implementing *dynamic cumulative frequency tables*. Suppose we have[29] test scores of `m = 11` students `f = {2,4,5,5,6,6,6,7,7,8,9}` where the test scores are *integer values* ranging from `[1..10]`. Table 2.1 shows the frequency of each individual test score ∈ `[1..10]` and the cumulative frequency of test scores ranging from `[1..i]` denoted by `cf[i]`—that is, the sum of the frequencies of test scores `1, 2, ..., i`.

Index/ Score	Frequency f	Cumulative Frequency cf	Short Comment
0	-	-	Index 0 is ignored (as the sentinel value).
1	0	0	`cf[1] = f[1] = 0.`
2	1	1	`cf[2] = f[1] + f[2] = 0 + 1 = 1.`
3	0	1	`cf[3] = f[1] + f[2] + f[3] = 0 + 1 + 0 = 1.`
4	1	2	`cf[4] = cf[3] + f[4] = 1 + 1 = 2.`
5	2	4	`cf[5] = cf[4] + f[5] = 2 + 2 = 4.`
6	3	7	`cf[6] = cf[5] + f[6] = 4 + 3 = 7.`
7	2	9	`cf[7] = cf[6] + f[7] = 7 + 2 = 9.`
8	1	10	`cf[8] = cf[7] + f[8] = 9 + 1 = 10.`
9	1	11	`cf[9] = cf[8] + f[9] = 10 + 1 = 11.`
10	0	11	`cf[10] = cf[9] + f[10] = 11 + 0 = 11.`

Table 2.1: Example of a Cumulative Frequency Table

The cumulative frequency table can also be used as a solution to the Range Sum Query (RSQ) problem mentioned in **Exercise 2.4.3.2***. It stores `RSQ(1,` i`)` $\forall i \in$ `[1..n]` where `n` is the largest integer index/score[30]. In the example above, we have `n = 10`, `RSQ(1, 1) = 0`, `RSQ(1, 2) = 1`, ..., `RSQ(1, 6) = 7`, ..., `RSQ(1, 8) = 10`, ..., and `RSQ(1, 10) = 11`. We can then obtain the answer to the RSQ for an arbitrary range `RSQ(i, j)` when $i \neq 1$ by subtracting `RSQ(1, j) - RSQ(1, i - 1)`. For example, `RSQ(4, 6) = RSQ(1, 6) - RSQ(1, 3) = 7 - 1 = 6`.

If the frequencies are *static*, then the cumulative frequency table as in Table 2.1 can be computed efficiently with a simple $O(n)$ loop. First, set `cf[1] = f[1]`. Then, for `i` ∈ `[2..n]`, compute `cf[i] = cf[i - 1] + f[i]`. This will be discussed further in Section 3.5.2. However, when the frequencies are frequently updated (increased or decreased) and the RSQs are frequently asked afterwards, it is better to use a *dynamic* data structure.

Instead of using a Segment Tree to implement a *dynamic* cumulative frequency table, we can implement the *far simpler* Fenwick Tree instead (compare the source code for both implementations, provided in this section and in the previous Section 2.4.3). This is perhaps one of the reasons why the Fenwick Tree is currently included in the IOI syllabus [20]. Fenwick Tree operations are also extremely efficient as they use fast bit manipulation techniques (see Section 2.2).

In this section, we will use the function `LSOne(i)` (which is actually `(i & (-i))`) extensively, naming it to match its usage in the original paper [18]. In Section 2.2, we have seen that the operation `(i & (-i))` produces the first Least Significant One-bit in `i`.

[29] The test scores are shown in sorted order for simplicity, they do not have to be sorted.

[30] Please differentiate `m` = the number of data points and `n` = the largest integer value among the `m` data points. The meaning of `n` in Fenwick Tree is a bit different compared to other data structures in this book.

The Fenwick Tree is typically implemented as an array (we use a `vector` for size flexibility). The Fenwick Tree is a tree that is indexed by the *bits* of its *integer* keys. These integer keys fall within the fixed range `[1..n]`—skipping[31] index 0. In a programming contest environment, `n` can approach $\approx 1M$ so that the Fenwick Tree covers the range `[1..1`M`]`—large enough for many practical (contest) problems. In Table 2.1 above, the scores `[1..10]` are the integer keys in the corresponding array with size `n = 10` and `m = 11` data points.

Let the name of the Fenwick Tree array be `ft`. Then, the element at index `i` is responsible for elements in the range `[i-LSOne(i)+1..i]` and `ft[i]` stores the cumulative frequency of elements {`i-LSOne(i)+1, i-LSOne(i)+2, i-LSOne(i)+3, .., i`}. In Figure 2.11, the value of `ft[i]` is shown in the circle above index `i` and the range `[i-LSOne(i)+1..i]` is shown as a circle and a bar (if the range spans more than one index) above index `i`. We can see that `ft[4] = 2` is responsible for range `[4-4+1..4] = [1..4]`, `ft[6] = 5` is responsible for range `[6-2+1..6] = [5..6]`, `ft[7] = 2` is responsible for range `[7-1+1..7] = [7..7]`, `ft[8] = 10` is responsible for range `[8-8+1..8] = [1..8]` etc[32].

With such an arrangement, if we want to obtain the cumulative frequency between `[1..b]`, i.e. `rsq(b)`, we simply add `ft[b]`, `ft[b']`, `ft[b'']`, ... until index `b`^{i} is 0. This sequence of indices is obtained via subtracting the Least Significant One-bit via the bit manipulation expression: `b' = b - LSOne(b)`. Iteration of this bit manipulation effectively *strips off* the least significant one-bit of `b` at each step. As an integer `b` only has $O(\log b)$ bits, `rsq(b)` runs in $O(\log n)$ time when `b = n`. In Figure 2.11, `rsq(6) = ft[6] + ft[4] = 5 + 2 = 7`. Notice that indices 4 and 6 are responsible for range `[1..4]` and `[5..6]`, respectively. By combining them, we account for the entire range of `[1..6]`. The indices 6, 4, and 0 are related in their binary form: `b` $= 6_{10} = (1\underline{1}0)_2$ can be transformed to `b'` $= 4_{10} = (\underline{1}00)_2$ and subsequently to `b''` $= 0_{10} = (000)_2$.

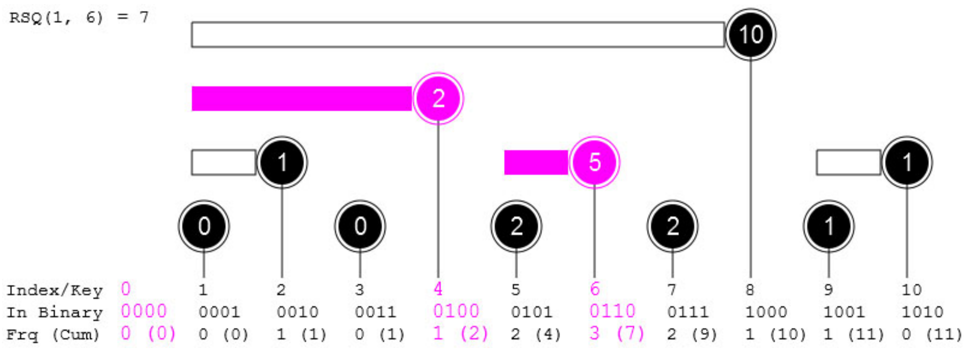

RSQ(1, 6) = 7
10
2
1
5
1
0
0
2
2
1
Index/Key 0 1 2 3 4 5 6 7 8 9 10
In Binary 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010
Frq (Cum) 0 (0) 0 (0) 1 (1) 0 (1) 1 (2) 2 (4) 3 (7) 2 (9) 1 (10) 1 (11) 0 (11)

Figure 2.11: Example of `rsq(6)`

With `rsq(b)` available, obtaining the cumulative frequency between two indices `[a..b]` where `a != 1` is simple, just evaluate `rsq(a, b) = rsq(b) - rsq(a - 1)`. For example, if we want to compute `rsq(4, 6)`, we can simply return `rsq(6) - rsq(3) = (5+2) - (0+1) = 7 - 1 = 6`. Again, this operation runs in $O(2 \times \log b) \approx O(\log n)$ time when `b = n`. Figure 2.12 displays the value of `rsq(3)`.

When updating the value of the element at index `k` by adjusting its value by `v` (note that `v` can be either positive or negative), i.e. calling `adjust(k, v)`, we have to update `ft[k]`, `ft[k']`, `ft[k'']`, ... until index `k`^{i} exceeds n. This sequence of indices are obtained

[31]We have chosen to follow the original implementation by [18] that ignores index 0 to facilitate an easier understanding of the bit manipulation operations of Fenwick Tree. Note that index 0 has no bit turned on. Thus, the operation `i +/- LSOne(i)` simply returns `i` when `i = 0`. Index 0 is also used as the terminating condition in the `rsq` function.

[32]In this book, we will not detail why this arrangement works and will instead show that it allows for efficient $O(\log n)$ update and RSQ operations. Interested readers are advised to read [18].

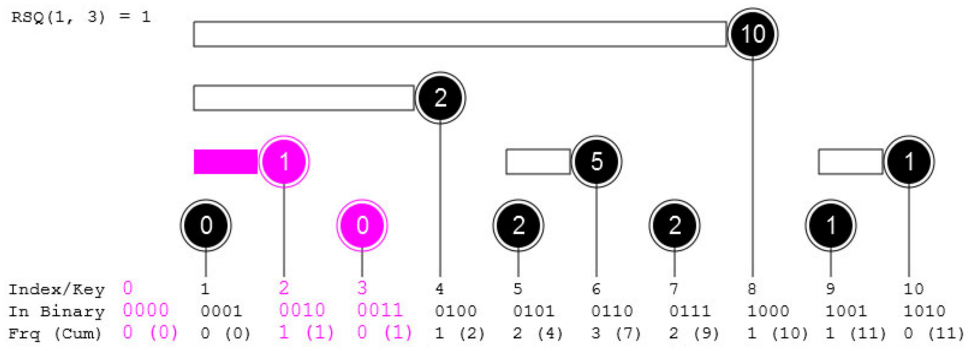

RSQ(1, 3) = 1
10
2
1
5
1
0
0
2
2
1
Index/Key 0 1 2 3 4 5 6 7 8 9 10
In Binary 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010
Frq (Cum) 0 (0) 0 (0) 1 (1) 0 (1) 1 (2) 2 (4) 3 (7) 2 (9) 1 (10) 1 (11) 0 (11)

Figure 2.12: Example of `rsq(3)`

via this similar iterative bit manipulation expression: `k' = k + LSOne(k)`. Starting from any integer `k`, the operation `adjust(k, v)` will take at most $O(\log n)$ steps until `k` $>$ `n`. In Figure 2.13, `adjust(5, 1)` will affect (add +1 to) `ft[k]` at indices `k = ` 5_{10} `= (101)`$_2$, `k' = (101)`$_2$ `+ (00`$\underline{1}$`)`$_2$ `= (110)`$_2$ `=` 6_{10}, and `k'' = (110)`$_2$ `+ (0`$\underline{1}$`0)`$_2$ `= (1000)`$_2$ `=` 8_{10} via the expression given above. Notice that if you project a line upwards from index 5 in Figure 2.13, you will see that the line indeed *intersects* the ranges under the responsibility of index 5, index 6, and index 8.

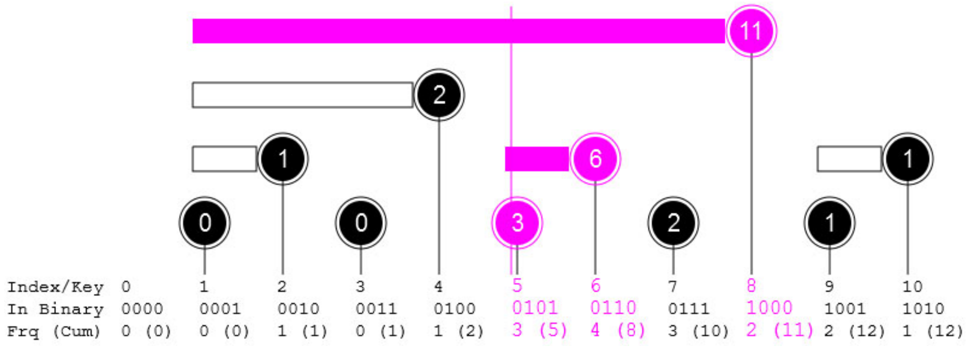

11
2
1
6
1
0
0
3
2
1
Index/Key 0 1 2 3 4 5 6 7 8 9 10
In Binary 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010
Frq (Cum) 0 (0) 0 (0) 1 (1) 0 (1) 1 (2) 3 (5) 4 (8) 3 (10) 2 (11) 2 (12) 1 (12)

Figure 2.13: Example of `adjust(5, 1)`

In summary, Fenwick Tree supports both RSQ and update operations in just $O(n)$ space and $O(\log n)$ time given a set of m integer keys that ranges from `[1..`n`]`. This makes Fenwick Tree an ideal data structure for solving *dynamic* RSQ problems on with discrete arrays (the *static* RSQ problem can be solved with simple $O(n)$ pre-processing and $O(1)$ per query as shown earlier). Our *short* C++ implementation of a basic Fenwick Tree is shown below.

```
class FenwickTree {
private: vi ft;               // recall that vi is: typedef vector<int> vi;
public: FenwickTree(int n) { ft.assign(n + 1, 0); }    // init n + 1 zeroes
  int rsq(int b) {                                     // returns RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }                              // note: LSOne(S) (S & (-S))
  int rsq(int a, int b) {                              // returns RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }
  // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
  void adjust(int k, int v) {                    // note: n = ft.size() - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};
```

```
int main() {
  int f[] = { 2,4,5,5,6,6,6,7,7,8,9 };                     // m = 11 scores
  FenwickTree ft(10);          // declare a Fenwick Tree for range [1..10]
      // insert these scores manually one by one into an empty Fenwick Tree
  for (int i = 0; i < 11; i++) ft.adjust(f[i], 1);    // this is O(k log n)
  printf("%d\n", ft.rsq(1, 1));  // 0 => ft[1] = 0
  printf("%d\n", ft.rsq(1, 2));  // 1 => ft[2] = 1
  printf("%d\n", ft.rsq(1, 6));  // 7 => ft[6] + ft[4] = 5 + 2 = 7
  printf("%d\n", ft.rsq(1, 10)); // 11 => ft[10] + ft[8] = 1 + 10 = 11
  printf("%d\n", ft.rsq(3, 6));  // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
  ft.adjust(5, 2);               // update demo
  printf("%d\n", ft.rsq(1, 10)); // now 13
} // return 0;
```

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/bit.html`

Source code: `ch2_10_fenwicktree_ds.cpp/java`

Exercise 2.4.4.1: Just a simple exercise of the two basic bit-manipulation operations used in the Fenwick Tree: What are the values of `90 - LSOne(90)` and `90 + LSOne(90)`?

Exercise 2.4.4.2: What if the problem that you want to solve includes an element at integer key 0? Recall that the standard integer key range in our library code is is `[1..n]` and that this implementation cannot use index 0 since it is used as the terminating condition of `rsq`.

Exercise 2.4.4.3: What if the problem that you want to solve uses non-integer keys? For example, what if the test scores shown in Table 2.1 above are `f = {5.5, 7.5, 8.0, 10.0}` (i.e. allowing either a 0 or a 5 after the decimal place)? What if the test scores are `f = {5.53, 7.57, 8.10, 9.91}` (i.e. allowing for two digits after the decimal point)?

Exercise 2.4.4.4: The Fenwick Tree supports an additional operation that we have decided to leave as an exercise to the reader: Find the smallest index with a given cumulative frequency. For example, we may need to determine the minimum index/score `i` in Table 2.1 such that there are at least 7 students covered in the range `[1..i]` (index/score 6 in this case). Implement this feature.

Exercise 2.4.4.5*: Solve this dynamic RSQ problem: UVa 12086 - Potentiometers using *both* a Segment Tree and Fenwick Tree. Which solution is easier to produce in this case? Also see Table 2.2 for a comparison between these two data structures.

Exercise 2.4.4.6*: Extend the 1D Fenwick Tree to 2D!

Exercise 2.4.4.7*: Fenwick Trees are normally used for point update and range (sum) query. Show how to use a Fenwick Tree for *range update* and point queries. For example, given lots of intervals with small ranges (from 1 to at most 1 million) determine the number of intervals encompassing index i.

Profile of Data Structure Inventors

Peter M. Fenwick is a Honorary Associate Professor in the University of Auckland. He invented the Binary Indexed Tree in 1994 [18] as "cumulative frequency tables of arithmetic compression". The BIT has since been included in the IOI syllabus [20] and used in many contest problems for its efficient yet easy to implement data structure.

Feature	Segment Tree	Fenwick Tree
Build Tree from Array	$O(n)$	$O(m \log n)$
Dynamic RMin/MaxQ	OK	Very limited
Dynamic RSQ	OK	OK
Query Complexity	$O(\log n)$	$O(\log n)$
Point Update Complexity	$O(\log n)$	$O(\log n)$
Length of Code	Longer	Shorter

Table 2.2: Comparison Between Segment Tree and Fenwick Tree

Programming exercises that use the data structures discussed and implemented:

- Graph Data Structures Problems
  1. **UVa 00599 - The Forrest for the Trees *** ($v-e$ = number of connected components, keep a `bitset` of size 26 to count the number of vertices that have some edge. Note: Also solvable with Union-Find)
  2. **UVa 10895 - Matrix Transpose *** (transpose adjacency list)
  3. UVa 10928 - My Dear Neighbours (counting out degrees)
  4. UVa 11550 - Demanding Dilemma (graph representation, incidence matrix)
  5. **UVa 11991 - Easy Problem from ... *** (use the idea of an Adj List)

  Also see: More graph problems in Chapter 4
- Union-Find Disjoint Sets
  1. **UVa 00793 - Network Connections *** (trivial; application of disjoint sets)
  2. UVa 01197 - The Suspects (LA 2817, Kaohsiung03, Connected Components)
  3. UVa 10158 - War (advanced usage of disjoint sets with a nice twist; memorize list of enemies)
  4. UVa 10227 - Forests (merge two disjoint sets if they are consistent)
  5. **UVa 10507 - Waking up brain *** (disjoint sets simplifies this problem)
  6. UVa 10583 - Ubiquitous Religions (count disjoint sets after all unions)
  7. UVa 10608 - Friends (find the set with the largest element)
  8. UVa 10685 - Nature (find the set with the largest element)
  9. **UVa 11503 - Virtual Friends *** (maintain set attribute (size) in rep item)
  10. UVa 11690 - Money Matters (check if total money from each member is 0)
- Tree-related Data Structures
  1. UVa 00297 - Quadtrees (simple quadtree problem)
  2. UVa 01232 - SKYLINE (LA 4108, Singapore07, a simple problem if input size is small; but since $n \leq 100000$, we have to use a Segment Tree; note that this problem is not about RSQ/RMQ)
  3. **UVa 11235 - Frequent Values *** (range maximum query)
  4. UVa 11297 - Census (Quad Tree with updates or use 2D segment tree)
  5. UVa 11350 - Stern-Brocot Tree (simple tree data structure question)
  6. **UVa 11402 - Ahoy, Pirates *** (segment tree with *lazy* updates)
  7. UVa 12086 - Potentiometers (LA 2191, Dhaka06; pure dynamic range sum query problem; solvable with Fenwick Tree or Segment Tree)
  8. ***UVa 12532 - Interval Product **** (clever usage of Fenwick/Segment Tree)

  Also see: DS as part of the solution of harder problems in Chapter 8

2.5 Solution to Non-Starred Exercises

Exercise 2.2.1*: Sub-question 1: First, sort S in $O(n \log n)$ and then do an $O(n)$ linear scan starting from the second element to check if an integer and the previous integer are the same (also read the solution for **Exercise 1.2.10**, task 4). Sub-question 6: Read the opening paragraph of Chapter 3 and the detailed discussion in Section 9.29. Solutions for the other sub-questions are not shown.

Exercise 2.2.2: The answers (except sub-question 7):

1. $S \mathbin{\&} (N-1)$
2. $(S \mathbin{\&} (S-1)) == 0$
3. $S \mathbin{\&} (S-1)$
4. $S \parallel (S+1)$
5. $S \mathbin{\&} (S+1)$
6. $S \parallel (S-1)$

Exercise 2.3.1: Since the collection is dynamic, we will encounter frequent insertion and deletion queries. An insertion can potentially change the sort order. If we store the information in a static array, we will have to use one $O(n)$ iteration of an insertion sort after each insertion and deletion (to close the gap in the array). This is inefficient!

Exercise 2.3.2:

1. `search(71)`: root (15) $\rightarrow$ 23 $\rightarrow$ 71 (found)
   `search(7)`: root (15) $\rightarrow$ 6 $\rightarrow$ 7 (found)
   `search(22)`: root (15) $\rightarrow$ 23 $\rightarrow$ empty left subtree (not found).
2. We will eventually have the same BST as in Figure 2.2.
3. To find the min/max element, we can start from root and keep going left/right until we encounter a vertex with no left/right subtrees respectively. That vertex is the answer.
4. We will obtain the sorted output: 4, 5, 6, 7, 15, 23, 50, 71. See Section 4.7.2 if you are not familiar with the inorder tree traversal algorithm.
5. `successor(23)`: Find the minimum element of the subtree rooted at the right of 23, which is the subtree rooted at 71. The answer is 50.
   `successor(7)`: 7 has no right subtree, so 7 must be the maximum of a certain subtree. That subtree is the subtree rooted at 6. The parent of 6 is 15 and 6 is the left subtree of 15. By the BST property, 15 must be the successor of 7.
   `successor(71)`: 71 is the largest element and has no successor.
   Note: The algorithm to find the predecessor of a node is similar.
6. `delete(5)`: We simply remove 5, which is a leaf, from the BST
   `delete(71)`: As 71 is an internal vertex with one child, we cannot simply delete 71 as doing so will disconnect the BST into *two* components. We can instead reshuffle the subtree rooted at the parent of 71 (which is 23), causing 23 to has 50 as its right child.

7. `delete(15)`: As 15 is a vertex with two children, we cannot simply delete 15 as doing so will disconnect the BST into *three* components. To deal with this issue, we need to find the successor of 15 (which is 23) and use the successor to replace 15. We then delete the old 23 from the BST (not a problem now). As a note, we can also use predecessor(key) instead of successor(key) during delete(key) for the case when the key has two children.

Exercise 2.3.3*: For Sub-task 1, we run inorder traversal in $O(n)$ and see if the values are sorted. Solutions to other sub-tasks are not shown.

Exercise 2.3.6: The answers:

1. `Insert(26)`: Insert 26 as the left subtree of 3, swap 26 with 3, then swap 26 with 19 and stop. The Max Heap array A now contains {-, 90, 26, 36, 17, 19, 25, 1, 2, 7, 3}.

2. `ExtractMax()`: Swap 90 (maximum element which will be reported after we fix the Max Heap property) with 3 (the current bottom-most right-most leaf/the last item in the Max Heap), swap 3 with 36, swap 3 with 25 and stop. The Max Heap array A now contains {-, 36, 26, 25, 17, 19, 3, 1, 2, 7}.

Exercise 2.3.7: Yes, check that all indices (vertices) satisfy the Max Heap property.

Exercise 2.3.16: Use the C++ STL `set` (or Java `TreeSet`) as it is a balanced BST that supports $O(\log n)$ dynamic insertions and deletions. We can use the inorder traversal to print the data in the BST in sorted order (simply use C++ `iterator`s or Java `Iterator`s).

Exercise 2.3.17: Use the C++ STL `map` (Java `TreeMap`) and a counter variable. A hash table is also a possible solution but not necessary for programming contests. This trick is quite frequently used in various (contest) problems. Example usage:

```
char str[1000];
map<string, int> mapper;
int i, idx;
for (i = idx = 0; i < M; i++) {                          // idx starts from 0
  scanf("%s", &str);
  if (mapper.find(str) == mapper.end())    // if this is the first encounter
  // alternatively, we can also test if mapper.count(str) is greater than 0
    mapper[str] = idx++;       // give str the current idx and increase idx
}
```

Exercise 2.4.1.3: The graph is undirected.

Exercise 2.4.1.4*: Subtask 1: To count the number of vertices of a graph: Adjacency Matrix/Adjacency List → report the number of rows; Edge List → count the number of distinct vertices in all edges. To count the number of edges of a graph: Adjacency Matrix → sum the number of non-zero entries in every row; Adjacency List → sum the length of all the lists; Edge List → simply report the number of rows. Solutions to other sub-tasks are not shown.

Exercise 2.4.2.1: For `int numDisjointSets()`, use an additional integer counter `numSets`. Initially, during `UnionFind(N)`, set `numSets = N`. Then, during `unionSet(i, j)`, decrease `numSets` by one if `isSameSet(i, j)` returns false. Now, `int numDisjointSets()` can simply return the value of `numSets`.

For `int sizeOfSet(int i)`, we use another `vi setSize(N)` initialized to all ones (each set has only one element). During `unionSet(i, j)`, update the `setSize` array by performing `setSize[find(j)] += setSize[find(i)]` (or the other way around depending on rank) if `isSameSet(i, j)` returns false. Now `int sizeOfSet(int i)` can simply return the value of `setSize[find(i)];`

These two variants have been implemented in `ch2_08_unionfind_ds.cpp/java`.

Exercise 2.4.3.3: `RSQ(1, 7) = 167` and `RSQ(3, 8) = 139`; No, using a Segment Tree is overkill. There is a simple DP solution that uses an $O(n)$ pre-processing step and takes $O(1)$ time per RSQ (see Section 9.33).

Exercise 2.4.4.1: `90 - LSOne(90) =` $(1011010)_2$ `-` $(10)_2$ `=` $(1011000)_2$ `= 88` and
`90 + LSOne(90) =` $(1011010)_2$ `+` $(10)_2$ `=` $(1011100)_2$ `= 92`.

Exercise 2.4.4.2: Simple: shift all indices by one. Index $i$in the 1-based Fenwick Tree now refers to index $i - 1$ in the actual problem.

Exercise 2.4.4.3: Simple: convert the floating point numbers into integers. For the first task, we can multiply every number by two. For the second case, we can multiply all numbers by one hundred.

Exercise 2.4.4.4: The cumulative frequency is sorted, thus we can use a *binary search*. Study the 'binary search for the answer' technique discussed in Section 3.3. The resulting time complexity is $O(\log^2 n)$.

2.6 Chapter Notes

The basic data structures mentioned in Section 2.2-2.3 can be found in almost every data structure and algorithm textbook. References to the C++/Java built-in libraries are available online at: `www.cppreference.com` and `java.sun.com/javase/7/docs/api`. Note that although access to these reference websites are usually provided in programming contests, we suggest that you try to master the syntax of the most common library operations to minimize coding time during actual contests!

One exception is perhaps the *lightweight set of Boolean* (a.k.a bitmask). This *unusual* technique is not commonly taught in data structure and algorithm classes, but it is quite important for competitive programmers as it allows for significant speedups if applied to certain problems. This data structure appears in various places throughout this book, e.g. in some iterative brute force and optimized backtracking routines (Section 3.2.2 and Section 8.2.1), DP TSP (Section 3.5.2), DP with bitmask (Section 8.3.1). All of them use bitmasks instead of `vector<boolean>` or `bitset<size>` due to its efficiency. Interested readers are encouraged to read the book "Hacker's Delight" [69] that discusses bit manipulation in further detail.

Extra references for the data structures mentioned in Section 2.4 are as follows. For Graphs, see [58] and Chapters 22-26 of [7]. For Union-Find Disjoint Sets, see Chapter 21 of [7]. For Segment Trees and other geometric data structures, see [9]. For the Fenwick Tree, see [30]. We remark that all our implementation of data structures discussed in Section 2.4 avoid the usage of pointers. We use either arrays or vectors.

With more experience and by reading the source code we have provided, you can master more tricks in the application of these data structures. Please spend time exploring the source code provided with this book at `sites.google.com/site/stevenhalim/home/material`.

There are few more data structures discussed in this book—string-specific data structures (**Suffix Trie/Tree/Array**) are discussed in Section 6.6. Yet, there are still many other data structures that we cannot cover in this book. If you want to do better in programming contests, please research data structure techniques beyond what we have presented in this book. For example, **AVL Trees**, **Red Black Trees**, or even **Splay Trees** are useful for certain problems that require you to implement and augment (add more data to) balanced BSTs (see Section 9.29). **Interval Tree**s (which are similar to Segment Trees) and **Quad Tree**s (for partitioning 2D space) are useful to know as their underlying concepts may help you to solve certain contest problems.

Notice that many of the efficient data structures discussed in this book exhibit the 'Divide and Conquer' strategy (discussed in Section 3.3).

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	12	18 (+50%)	35 (+94%)
Written Exercises	5	12 (+140%)	14+27*=41 (+242%)
Programming Exercises	43	124 (+188%)	132 (+6%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
2.2	**Linear DS**	79	60%	5%
2.3	**Non-Linear DS**	30	23%	2%
2.4	Our-own Libraries	23	17%	1%