Chapter

5 Primitive Types

Representation is the essence of programming.

— "*The Mythical Man Month*,"
F. P. Brooks, 1975

A program updates variables in memory according to its instructions. Variables come in types—a type is a classification of data that spells out possible values for that type and the operations that can be performed on it. A type can be provided by the language or defined by the programmer. Many languages provide types for Boolean, integer, character and floating point data. Often, there are multiple integer and floating point types, depending on signedness and precision. The width of these types is the number of bits of storage a corresponding variable takes in memory. For example, most implementations of C++ use 32 or 64 bits for an `int`. In Java an `int` is always 32 bits.

Primitive types boot camp

Writing a program to count the number of bits that are set to 1 in an integer is a good way to get up to speed with primitive types. The following program tests bits one-at-a-time starting with the least-significant bit. It illustrates shifting and masking; it also shows how to avoid hard-coding the size of the integer word.

```
short CountBits(unsigned int x) {
  short num_bits = 0;
  while (x) {
    num_bits += x & 1;
    x >>= 1;
  }
  return num_bits;
}
```

Since we perform $O(1)$ computation per bit, the time complexity is $O(n)$, where n is the number of bits in the integer word. Note that, by definition, the time complexity is for the worst case input, which for this example is the word $(111\ldots11)_2$. In the best case, the time complexity is $O(1)$, e.g., if the input is 0. The techniques in Solution 5.1 on the facing page, which is concerned with counting the number of bits modulo 2, i.e., the parity, can be used to improve the performance of the program given above.

Know your primitive types

You should know the primitive types very intimately, e.g., sizes, ranges, signedness properties, and operators. You should also know the utility methods for primitive types in `cmath`. The `random` library is also very helpful, especially when writing tests.

- Be very familiar with the bit-wise operators such as `6&4`, `1|2`, `8>>1`, `1<<10`, `~0`, `15^x`.

Be very comfortable with the **bitwise operators**, particularly XOR. [Problem 5.2]

Understand how to use **masks** and create them in an **machine independent** way. [Problem 5.6]

Know fast ways to **clear the lowermost set bit** (and by extension, set the lowermost 0, get its index, etc.) [Problem 5.1]

Understand **signedness** and its implications to **shifting**. [Problem 5.5]

Consider using a **cache** to accelerate operations by using it to brute-force small inputs. [Problem 5.3]

Be aware that **commutativity** and **associativity** can be used to perform operations in **parallel** and **reorder** operations. [Problem 5.1]

- Know the constants denoting the maximum and minimum values for numeric types, etc., e.g., `numeric_limits<int>::min()`, `numeric_limits<float>.max()`, `numeric_limits<double>::infinity()`.
- Take extra care when comparing floating point values. Take a look of Solution 12.5 on Page 176 for an example.
- Key methods in `cmath` are `abs(-34)`, `fabs(-3.14)`, `ceil(2.17)`, `floor(3.14)`, `min(x, -4)`, `max(3.14, y)`, `pow(2.71, 3.14)`, `log(7.12)`, and `sqrt(225)`.
- Know how to interconvert integers, characters, and strings, e.g., `x - '0'` to convert a digit character to an integer.
- Key methods in `random` are `uniform_int_distribution<> dis(1, 6)` (which returns an integer value in $[1, 6]$), `uniform_real_distribution<double> dis(1.3, 2.9)` (which returns a floating point number in $[1.3, 2.9]$), `generate_canonical<double, 10>` (which returns a value in $[0, 1)$).

5.1 Computing the parity of a word

The parity of a binary word is 1 if the number of 1s in the word is odd; otherwise, it is 0. For example, the parity of 1011 is 1, and the parity of 10001000 is 0. Parity checks are used to detect single bit errors in data storage and communication. It is fairly straightforward to write code that computes the parity of a single 64-bit word.

How would you compute the parity of a very large number of 64-bit words?

Hint: Use a lookup table, but don't use 2^{64} entries!

Solution: The brute-force algorithm iteratively tests the value of each bit while tracking the number of 1s seen so far. Since we only care if the number of 1s is even or odd, we can store the number modulo 2.

```
short Parity(unsigned long x) {
  short result = 0;
  while (x) {
    result ^= (x & 1);
    x >>= 1;
```

Chapter

6 Arrays

The machine can alter the scanned symbol and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine.

— "*Intelligent Machinery*,"
A. M. Turing, 1948

The simplest data structure is the array, which is a contiguous block of memory. It is usually used to represent sequences. Given an array A, $A[i]$ denotes the $(i+1)$th object stored in the array. Retrieving and updating $A[i]$ takes $O(1)$ time. Insertion into a full array can be handled by resizing, i.e., allocating a new array with additional memory and copying over the entries from the original array. This increases the worst-case time of insertion, but if the new array has, for example, a constant factor larger than the original array, the average time for insertion is constant since resizing is infrequent. Deleting an element from an array entails moving all successive elements one over to the left to fill the vacated space. For example, if the array is $\langle 2, 3, 5, 7, 9, 11, 13, 17\rangle$, then deleting the element at index 4 results in the array $\langle 2, 3, 5, 7, 11, 13, 17, 0\rangle$. (We do not care about the last value.) The time complexity to delete the element at index i from an array of length n is $O(n-i)$.

Array boot camp

The following problem gives good insight into working with arrays: Your input is an array of integers, and you have to reorder its entries so that the even entries appear first. This is easy if you use $O(n)$ space, where n is the length of the array. However, you are required to solve it without allocating additional storage.

When working with arrays you should take advantage of the fact that you can operate efficiently on both ends. For this problem, we can partition the array into three subarrays: Even, Unclassified, and Odd, appearing in that order. Initially Even and Odd are empty, and Unclassified is the entire array. We iterate through Unclassified, moving its elements to the boundaries of the Even and Odd subarrays via swaps, thereby expanding Even and Odd, and shrinking Unclassified.

```
void EvenOdd(vector<int>* A_ptr) {
  vector<int>& A = *A_ptr;
  int next_even = 0, next_odd = A.size() - 1;
  while (next_even < next_odd) {
    if (A[next_even] % 2 == 0) {
      ++next_even;
    } else {
      swap(A[next_even], A[next_odd--]);
    }
  }
}
```

The additional space complexity is clearly $O(1)$—a couple of variables that hold indices, and a temporary variable for performing the swap. We do a constant amount of processing per entry, so the time complexity is $O(n)$.

Array problems often have simple brute-force solutions that use $O(n)$ space, but subtler solutions that **use the array itself** to **reduce space** complexity to $O(1)$. [Problem 6.1]

Filling an array from the front is slow, so see if it's possible to **write values from the back**. [Problem 6.2]

Instead of deleting an entry (which requires moving all entries to is right), consider **overwriting** it. [Problem 6.5]

When dealing with integers encoded by an array consider reversing the array so the **least-significant digit is the first entry**. [Problem 6.3]

Be comfortable with writing code that operates on **subarrays**. [Problem 6.10]

It's incredibly easy to make **off-by-1** errors when operating on arrays. [Problems 6.4 and 6.17]

Don't worry about preserving the **integrity** of the array (sortedness, keeping equal entries together, etc.) until it is time to return. [Problem 6.5]

An array can serve as a good data structure when you know the distribution of the elements in advance. For example, a Boolean array of length W is a good choice for representing a **subset of** $\{0, 1, \ldots, W-1\}$. (When using a Boolean array to represent a subset of $\{1, 2, 3, \ldots, n\}$, allocate an array of size $n+1$ to simplify indexing.) [Problem 6.8].

When operating on 2D arrays, **use parallel logic** for rows and for columns. [Problem 6.17]

Sometimes it's easier to **simulate the specification**, than to analytically solve for the result. For example, rather than writing a formula for the i-th entry in the spiral order for an $n \times n$ matrix, just compute the output from the beginning. [Problems 6.17 and 6.19]

Know your array libraries

You should know the C++ `array` and `vector` classes very intimately. Note that `array` is fixed-size, and `vector` is dynamically-resized. There are a number of very useful methods in the `algorithm` library.

- Know the syntax for allocating and instantiating an `array` or a `vector`, i.e., `array<int, 3> A = {1, 2, 3}`, `vector<int> A = {1, 2, 3}`. To construct a subarray from an array, you can use `vector<int> subarray_A(A.begin() + i, A.begin() + j)`—this sets `subarray_A` to be $A[i : j-1]$.
- Understand how to instantiate a 2D array—`array<array<int, 2>, 3> A = {{1, 2}, {3, 4}, {5, 6}}` and `vector<vector<int>> A = {{1, 2}, {3, 4}, {5, 6}}` both create an array which will hold three rows where each column holds two elements.

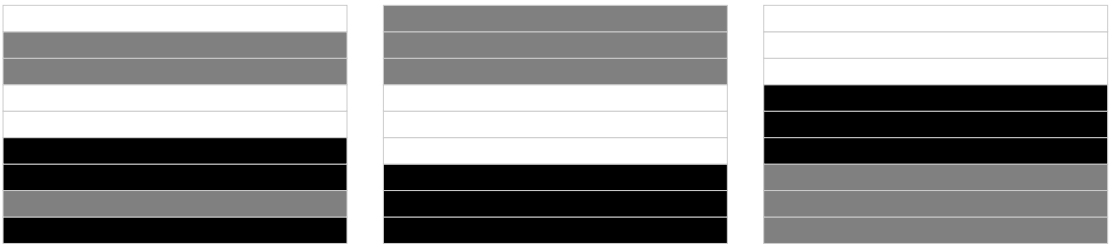
- Since `vector` is dynamically sizable, `push_back(42)` (or `emplace_back(42)`) are frequently used to add values to the end.
- Understand what "deep" means when checking equality of arrays, and hashing them.
- Key methods in `algorithms` include: `binary_search(A.begin(), A.end(), 42)`, `lower_bound(A.begin(), A.end(), 42)`, `upper_bound(A.begin(), A.end(), 42)`, `fill(A.begin(), A.end(), 42)`, `swap(x, y)`, `min_element(A.begin(), A.end())`, `max_element(A.begin(), A.end())`, `reverse(A.begin(), A.end())`, `rotate(A.begin(), A.begin() + shift, A.end())`, and `sort(A.begin(), A.end())`.
- Understand the variants of these methods, e.g., how to create a copy of a subarray.

6.1 The Dutch national flag problem

The quicksort algorithm for sorting arrays proceeds recursively—it selects an element (the "pivot"), reorders the array to make all the elements less than or equal to the pivot appear first, followed by all the elements greater than the pivot. The two subarrays are then sorted recursively.

Implemented naively, quicksort has large run times and deep function call stacks on arrays with many duplicates because the subarrays may differ greatly in size. One solution is to reorder the array so that all elements less than the pivot appear first, followed by elements equal to the pivot, followed by elements greater than the pivot. This is known as Dutch national flag partitioning, because the Dutch national flag consists of three horizontal bands, each in a different color.

As an example, assuming that black precedes white and white precedes gray, Figure 6.1(b) is a valid partitioning for Figure 6.1(a). If gray precedes black and black precedes white, Figure 6.1(c) is a valid partitioning for Figure 6.1(a).



(a) Before partitioning.

(b) A three-way partitioning resembling the Dutch national flag.

(c) Another three-way partitioning: the Russian national flag.

Figure 6.1: Illustrating the Dutch national flag problem.

Write a program that takes an array A and an index i into A, and rearranges the elements such that all elements less than $A[i]$ (the "pivot") appear first, followed by elements equal to the pivot, followed by elements greater than the pivot.

Hint: Think about the partition step in quicksort.

Solution: The problem is trivial to solve with $O(n)$ additional space, where n is the length of A. We form three lists, namely, elements less than the pivot, elements equal to the pivot, and elements greater than the pivot. Consequently, we write these values into A. The time complexity is $O(n)$.

We can avoid using $O(n)$ additional space at the cost of increased time complexity as follows. In the first stage, we iterate through A starting from index 0, then index 1, etc. In each iteration, we

Chapter

7 Strings

String pattern matching is an important problem that occurs in many areas of science and information processing. In computing, it occurs naturally as part of data processing, text editing, term rewriting, lexical analysis, and information retrieval.

— "*Algorithms For Finding Patterns in Strings*,"
A. V. Aho, 1990

Strings are ubiquitous in programming today—scripting, web development, and bioinformatics all make extensive use of strings. You should know how strings are represented in memory, and understand basic operations on strings such as comparison, copying, joining, splitting, matching, etc. We now present problems on strings which can be solved using elementary techniques. Advanced string processing algorithms often use hash tables (Chapter 13) and dynamic programming (Page 272).

Strings boot camp

A palindromic string is one which reads the same when it is reversed. The program below checks whether a string is palindromic. Rather than creating a new string for the reverse of the input string, it traverses the input string forwards and backwards, thereby saving space. Notice how it uniformly handles even and odd length strings.

```
bool IsPalindromic(const string& s) {
  for (int i = 0, j = s.size() - 1; i < j; ++i, --j) {
    if (s[i] != s[j]) {
      return false;
    }
  }
  return true;
}
```

The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the length of the string.

Similar to arrays, string problems often have simple brute-force solutions that use $O(n)$ space solution, but subtler solutions that uses the string itself to **reduce space complexity** to $O(1)$. [Problems 7.6 and 7.4]

Understand the **implications** of a string type that's **immutable**, e.g., the need to allocate a new string when concatenating immutable strings. Know **alternatives** to immutable strings, e.g., an array of characters or a `StringBuilder` in Java. [Problem 7.6]

Updating a mutable string from the front is slow, so see if it's possible to **write values from the back**. [Problem 7.4]

Know your string libraries

When manipulating C++ strings, you need to know the `string` class well.

- The basic methods are `append("Gauss")`, `push_back('c')`, `pop_back()`, `insert(s.begin() + shift, "Gauss")`, `substr(pos, len)`, and `compare("Gauss")`.
- Remember a string is organized like like an array. It performs well for operations from the back, e.g., `push_back('c')` and `pop_back()`, but poorly in the middle of a string, e.g., `insert(A.begin() + middle, "Gauss")`.
- The comparison operators <, <=, >, >=, and == can be applied to strings, with == testing logical equality, rather than pointer equality.

7.1 Interconvert strings and integers

A string is a sequence of characters. A string may encode an integer, e.g., "123" encodes 123. In this problem, you are to implement methods that take a string representing an integer and return the corresponding integer, and vice versa. Your code should handle negative integers. You cannot use library functions like `stoi` in C++ and `parseInt` in Java.

Implement string/integer inter-conversion functions.

Hint: Build the result one digit at a time.

Solution: Let's consider the integer to string problem first. If the number to convert is a single digit, i.e., it is between 0 and 9, the result is easy to compute: it is the string consisting of the single character encoding that digit.

If the number has more than one digit, it is natural to perform the conversion digit-by-digit. The key insight is that for any positive integer x, the least significant digit in the decimal representation of x is $x \mod 10$, and the remaining digits are $x/10$. This approach computes the digits in reverse order, e.g., if we begin with 423, we get 3 and are left with 42 to convert. Then we get 2, and are left with 4 to convert. Finally, we get 4 and there are no digits to convert. The natural algorithm would be to prepend digits to the partial result. However, adding a digit to the beginning of a string is expensive, since all remaining digit have to be moved. A more time efficient approach is to add each computed digit to the end, and then reverse the computed sequence.

If x is negative, we record that, negate x, and then add a '-' before reversing. If x is 0, our code breaks out of the iteration without writing any digits, in which case we need to explicitly set a 0.

To convert from a string to an integer we recall the basic working of a positional number system. A base-10 number $d_2d_1d_0$ encodes the number $10^2 \times d_2 + 10^1 \times d_1 + d_0$. A brute-force algorithm then is to begin with the rightmost digit, and iteratively add $10^i \times d_i$ to a cumulative sum. The efficient way to compute 10^{i+1} is to use the existing value 10^i and multiply that by 10.

A more elegant solution is to begin from the leftmost digit and with each succeeding digit, multiply the partial result by 10 and add that digit. For example, to convert "314" to an integer, we initial the partial result r to 0. In the first iteration, $r = 3$, in the second iteration $r = 3 \times 10 + 1 = 31$, and in the third iteration $r = 31 \times 10 + 4 = 314$, which is the final result.

Negative numbers are handled by recording the sign and negating the result.

```
string IntToString(int x) {
  bool is_negative = false;
  if (x < 0) {
    x = -x, is_negative = true;
  }
```

Chapter

8 Linked Lists

The S-expressions are formed according to the following recursive rules.

1. *The atomic symbols p_1, p_2, etc., are S-expressions.*
2. *A null expression $\wedge$ is also admitted.*
3. *If e is an S-expression so is (e).*
4. *If e_1 and e_2 are S-expressions so is (e_1, e_2).*

— "*Recursive Functions Of Symbolic Expressions*,"
J. McCarthy, 1959

A *singly linked list* is a data structure that contains a sequence of nodes such that each node contains an object and a reference to the next node in the list. The first node is referred to as the *head* and the last node is referred to as the *tail*; the tail's next field is null. The structure of a singly linked list is given in Figure 8.1. There are many variants of linked lists, e.g., in a *doubly linked list*, each node has a link to its predecessor; similarly, a sentinel node or a self-loop can be used instead of null. The structure of a doubly linked list is given in Figure 8.2.

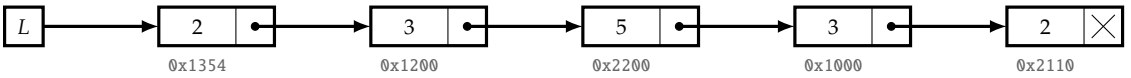


Figure 8.1: Example of a singly linked list. The number in hex below a node indicates the memory address of that node.

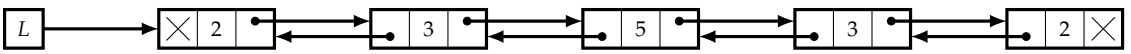


Figure 8.2: Example of a doubly linked list.

For all problems in this chapter, unless otherwise stated, each node has two entries—a data field, and a next field, which points to the next node in the list, with the next field of the last node being null. Its prototype is as follows:

```
template <typename T>
struct ListNode {
  T data;
  shared_ptr<ListNode<T>> next;
};
```

Linked lists boot camp

There are two types of list-related problems—those where you have to implement your own list, and those where you have to exploit the standard list library. We will review both these aspects here, starting with implementation, then moving on to list libraries.

Implementing a basic list API—search, insert, delete—for singly linked lists is an excellent way to become comfortable with lists.

- Search for a key.

```
shared_ptr<ListNode<int>> SearchList(shared_ptr<ListNode<int>> L, int key) {
  while (L && L->data != key) {
    L = L->next;
  }
  // If key was not present in the list, L will have become null.
  return L;
}
```

- Insert a new node after a specified node.

```
// Insert new_node after node.
void InsertAfter(const shared_ptr<ListNode<int>>& node,
                 const shared_ptr<ListNode<int>>& new_node) {
  new_node->next = node->next;
  node->next = new_node;
}
```

- Delete a node.

```
// Delete the node past this one. Assume node is not tail.
void DeleteAfter(const shared_ptr<ListNode<int>>& node) {
  node->next = node->next->next;
}
```

Insert and delete are local operations and have $O(1)$ time complexity. Search requires traversing the entire list, e.g., if the key is at the last node or is absent, so its time complexity is $O(n)$, where n is the number of nodes.

List problems often have a simple brute-force solution that uses $O(n)$ space, but a subtler solution that uses the **existing list nodes** to reduce space complexity to $O(1)$. [Problems 8.1 and 8.10]

Very often, a problem on lists is conceptually simple, and is more about **cleanly coding what's specified**, rather than designing an algorithm. [Problem 8.12]

Consider using a **dummy head** (sometimes referred to as a sentinel) to avoid having to check for empty lists. This simplifies code, and makes bugs less likely. [Problem 8.2]

It's easy to forget to **update next** (and previous for double linked list) for the head and tail. [Problem 8.10]

Algorithms operating on singly linked lists often benefit from using **two iterators**, one ahead of the other, or one advancing quicker than the other. [Problem 8.3]

Know your linked list libraries

We now review the standard linked list library, with the reminder that many interview problems that are directly concerned with lists require you to write your own list class. When manipulating C++ lists, you need to know the `list` and `forward_list` classes well. The `list` class is a doubly-linked list; `forward_list` is a singly-linked list.[1]

For doubly-linked lists, i.e., `list`, here are the functions you should know well.

[1] A singly-linked list is more space efficient than doubly-linked list because nodes do not contain a previous field. The trade-off is the inability to do bidirectional iteration.

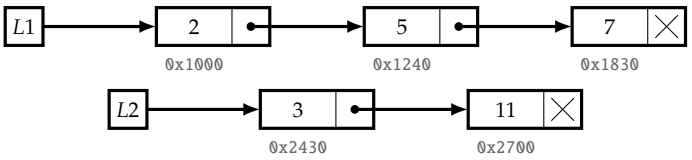
- The functions to insert and delete elements in list are `push_front(42)` (or `emplace_front(42)`), `pop_front()`, `push_back()` (or `emplace_back()`, and `pop_back()`).
- The `splice(L1.end(), L2)`, `reverse()`, and `sort()` functions are analogous to those on `forward_list`.

For singly-linked lists, i.e., `forward_list`, here are the functions you should know well.

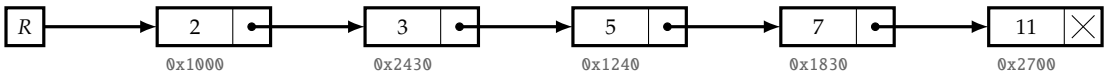
- The functions to insert and delete elements in list are `push_front(42)` (or `emplace_front(42)`), `pop_front()`, `insert_after(L.end(), 42)` (or `emplace_after(L.end(), 42)`, and `erase_after(A.begin())`).
- To transfer elements from list to another used `splice_after(L1.end(), L2)`.
- Reverse the order of the elements with `reverse()`.
- Use `sort()` to sort lists, and save yourself a great deal of pain.

8.1 Merge two sorted lists

Consider two singly linked lists in which each node holds a number. Assume the lists are sorted, i.e., numbers in the lists appear in ascending order within each list. The *merge* of the two lists is a list consisting of the nodes of the two lists in which numbers appear in ascending order. Merge is illustrated in Figure 8.3.



(a) Two sorted lists.



(b) The merge of the lists in (a).

Figure 8.3: Merging sorted lists.

Write a program that takes two lists, assumed to be sorted, and returns their merge. The only field your program can change in a node is its next field.

Hint: Two sorted arrays can be merged using two indices. For lists, take care when one iterator reaches the end.

Solution: A naive approach is to append the two lists together and sort the resulting list. The drawback of this approach is that it does not use the fact that the initial lists are sorted. The time complexity is that of sorting, which is $O((n + m)\log(n + m))$, where n and m are the lengths of each of the two input lists.

A better approach, in terms of time complexity, is to traverse the two lists, always choosing the node containing the smaller key to continue traversing from.

```
shared_ptr<ListNode<int>> MergeTwoSortedLists(shared_ptr<ListNode<int>> L1,
                                              shared_ptr<ListNode<int>> L2) {
  // Creates a placeholder for the result.
  shared_ptr<ListNode<int>> dummy_head(new ListNode<int>);
  auto tail = dummy_head;
```

Chapter

9 Stacks and Queues

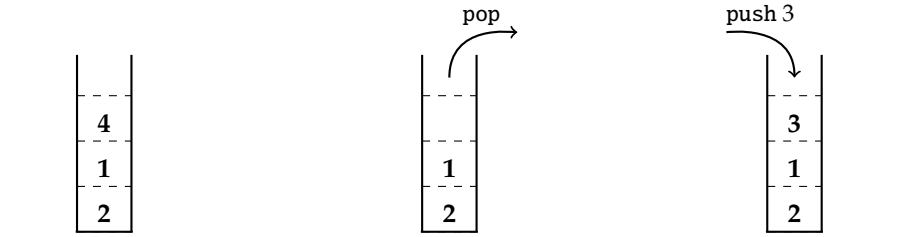
Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names . . .

— "*The Art of Computer Programming, Volume* 1,"
D. E. Knuth, 1997

Stacks

A *stack* supports two basic operations—push and pop. Elements are added (pushed) and removed (popped) in last-in, first-out order, as shown in Figure 9.1. If the stack is empty, pop typically returns null or throws an exception.

When the stack is implemented using a linked list these operations have $O(1)$ time complexity. If it is implemented using an array, there is maximum number of entries it can have—push and pop are still $O(1)$. If the array is dynamically resized, the amortized time for both push and pop is $O(1)$. A stack can support additional operations such as peek, which returns the top of the stack without popping it.



(a) Initial configuration. (b) Popping (a). (c) Pushing 3 on to (b).

Figure 9.1: Operations on a stack.

Stacks boot camp

The last-in, first-out semantics of a stack make it very useful for creating reverse iterators for sequences which are stored in a way that would make it difficult or impossible to step back from a given element. This a program uses a stack to print the entries of a singly-linked list in reverse order.

```
void PrintLinkedListInReverse(shared_ptr<ListNode<int>> head) {
  stack<int> nodes;
  while (head) {
    nodes.push(head->data);
```

```
    head = head->next;
  }
  while (!nodes.empty()) {
    cout << nodes.top() << endl;
    nodes.pop();
  }
}
```

The time and space complexity are $O(n)$, where n is the number of nodes in the list.

As an alternative, we could form the reverse of the list using Solution 8.2 on Page 103, iterate through the list printing entries, then perform another reverse to recover the list—this would have $O(n)$ time complexity and $O(1)$ space complexity.

Learn to recognize when the stack **LIFO** property is **applicable**. For example, **parsing** typically benefits from a stack. [Problems 9.2 and 9.6]

Consider **augmenting** the basic stack or queue data structure to support additional operations, such as finding the maximum element. [Problem 9.1]

Know your stack libraries

When manipulating C++ stacks, you need to know the `stack` class well. The key functions in the `stack` class are `top()`, `push(42)` (or `emplace(42)`), and `pop()`. When called from an empty stack, `top()` and `pop()` throw exceptions.

- `push(e)` pushes an element onto the stack. Not much can go wrong with a call to push.
- `top()` will retrieve, but does not remove, the element at the top of the stack.
- `pop()` will remove and the element at the top of the stack but does not return. To avoid the exception, first test with `empty()`.
- `empty()` tests if the stack is empty.

9.1 Implement a stack with max API

Design a stack that includes a max operation, in addition to push and pop. The max method should return the maximum value stored in the stack.

Hint: Use additional storage to track the maximum value.

Solution: The simplest way to implement a max operation is to consider each element in the stack, e.g., by iterating through the underlying array for an array-based stack. The time complexity is $O(n)$ and the space complexity is $O(1)$, where n is the number of elements currently in the stack.

The time complexity can be reduced to $O(\log n)$ using auxiliary data structures, specifically, a heap or a BST, and a hash table. The space complexity increases to $O(n)$ and the code is quite complex.

Suppose we use a single auxiliary variable, M, to record the element that is maximum in the stack. Updating M on pushes is easy: $M = \max(M, e)$, where e is the element being pushed. However, updating M on pop is very time consuming. If M is the element being popped, we have no way of knowing what the maximum remaining element is, and are forced to consider all the remaining elements.

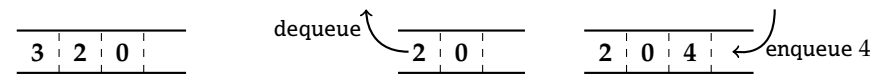
```
    candidates.emplace(BuildingWithHeight{building_idx++, building_height});
  }

  vector<int> buildings_with_sunset;
  while (!candidates.empty()) {
    buildings_with_sunset.emplace_back(candidates.top().id);
    candidates.pop();
  }
  return buildings_with_sunset;
}
```

Variant: Solve the problem subject to the same constraints when buildings are presented in west-to-east order.

Queues

A *queue* supports two basic operations—enqueue and dequeue. (If the queue is empty, dequeue typically returns null or throws an exception.) Elements are added (enqueued) and removed (dequeued) in first-in, first-out order. The most recently inserted element is referred to as the tail or back element, and the item that was inserted least recently is referred to as the head or front element.

A queue can be implemented using a linked list, in which case these operations have $O(1)$ time complexity. The queue API often includes other operations, e.g., a method that returns the item at the head of the queue without removing it, a method that returns the item at the tail of the queue without removing it, etc. A queue can also be implemented using an array; see Problem 9.8 on Page 130 for details.



(a) Initial configuration. (b) Queue (a) after dequeue. (c) Queue (b) after enqueuing 4.

Figure 9.4: Examples of enqueuing and dequeuing.

A *deque*, also sometimes called a double-ended queue, is a doubly linked list in which all insertions and deletions are from one of the two ends of the list, i.e., at the head or the tail. An insertion to the front is commonly called a push, and an insertion to the back is commonly called an inject. A deletion from the front is commonly called a pop, and a deletion from the back is commonly called an eject. (Different languages and libraries may have different nomenclature.)

Queues boot camp

In the following program, we implement the basic queue API—enqueue and dequeue—as well as a max-method, which returns the maximum element stored in the queue. The basic idea is to use composition: add a private field that references a library queue object, and forward existing methods (enqueue and dequeue in this case) to that object.

```
class Queue {
 public:
  void Enqueue(int x) { data_.emplace_back(x); }

  int Dequeue() {
```

```
    if (data_.empty()) {
      throw length_error("empty queue");
    }
    int val = data_.front();
    data_.pop_front();
    return val;
  }

  int Max() const {
    if (data_.empty()) {
      throw length_error("Cannot perform Max() on empty queue.");
    }
    return *max_element(data_.begin(), data_.end());
  }

 private:
  list<int> data_;
};
```

The time complexity of enqueue and dequeue are the same as that of the library queue, namely, $O(1)$. The time complexity of finding the maximum is $O(n)$, where n is the number of entries. In Solution 9.10 on Page 132 we show how to improve the time complexity of maximum to $O(1)$ with a more customized approach.

Learn to recognize when the queue FIFO property is applicable. For example, queues are ideal when order needs to be preserved. [Problem 9.7]

Know your queue libraries

When manipulating C++ queues, you need to know the `queue` class well. The key functions in the `queue` class are `front()`, `back()`, `push(42)` (or `emplace(42)`, and `pop()`). When called on an empty queue, `front()`, `back()`, and `pop()` throw exceptions.

- `push(e)` pushes an element onto the queue. Not much can go wrong with a call to push.
- `front()` will retrieve, but does not remove, the element at the front of the queue. Similarly, `back()` will retrieve, but also does not remove, the element at the back of the queue.
- `pop()` will remove and the element at the top of the queue but does not return.

`deque` implements the double-ended queue in C++, and `push_back(123)` (or `emplace_back(123)`), `push_front()` (or `emplace_front(123)`), `pop_back()`, `pop_front()`, `front()`, and `back()` are the key functions.

9.7 Compute binary tree nodes in order of increasing depth

Binary trees are formally defined in Chapter 10. In particular, each node in a binary tree has a depth, which is its distance from the root.

Given a binary tree, return an array consisting of the keys at the same level. Keys should appear in the order of the corresponding nodes' depths, breaking ties from left to right. For example, you should return $\langle\langle 314\rangle, \langle 6, 6\rangle, \langle 271, 561, 2, 271\rangle, \langle 28, 0, 3, 1, 28\rangle, \langle 17, 401, 257\rangle, \langle 641\rangle\rangle$ for the binary tree in Figure 10.1 on Page 135.

Hint: First think about solving this problem with a pair of queues.

Chapter

10 Binary Trees

The method of solution involves the development of a theory of finite automata operating on infinite trees.

— "*Decidability of Second Order Theories and Automata on Trees*," M. O. Rabin, 1969

A *binary tree* is a data structure that is useful for representing hierarchy. Formally, a binary tree is either empty, or a *root* node r together with a left binary tree and a right binary tree. The subtrees themselves are binary trees. The left binary tree is sometimes referred to as the *left subtree* of the root, and the right binary tree is referred to as the *right subtree* of the root.

Figure 10.1 gives a graphical representation of a binary tree. Node A is the root. Nodes B and I are the left and right children of A.

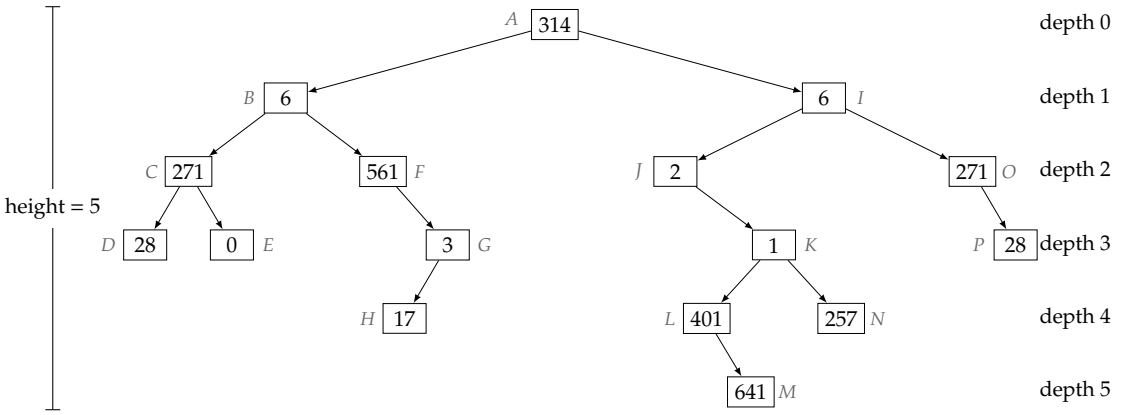


Figure 10.1: Example of a binary tree. The node depths range from 0 to 5. Node M has the highest depth (5) of any node in the tree, implying the height of the tree is 5.

Often the root stores additional data. Its prototype is listed as follows:

```
template <typename T>
struct BinaryTreeNode {
  T data;
  unique_ptr<BinaryTreeNode<T>> left, right;
};
```

Each node, except the root, is itself the root of a left subtree or a right subtree. If l is the root of p's left subtree, we will say l is the *left child* of p, and p is the *parent* of l; the notion of *right child* is similar. If a node is a left or a right child of p, we say it is a *child* of p. Note that with the exception of the root, every node has a unique parent. Usually, but not universally, the node object definition

includes a parent field (which is null for the root). Observe that for any node there exists a unique sequence of nodes from the root to that node with each node in the sequence being a child of the previous node. This sequence is sometimes referred to as the *search path* from the root to the node.

The parent-child relationship defines an ancestor-descendant relationship on nodes in a binary tree. Specifically, a node is an *ancestor* of d if it lies on the search path from the root to d. If a node is an ancestor of d, we say d is a *descendant* of that node. Our convention is that a node is an ancestor and descendant of itself. A node that has no descendants except for itself is called a *leaf*.

The *depth* of a node n is the number of nodes on the search path from the root to n, not including n itself. The *height* of a binary tree is the maximum depth of any node in that tree. A *level* of a tree is all nodes at the same depth. See Figure 10.1 on the preceding page for an example of the depth and height concepts.

As concrete examples of these concepts, consider the binary tree in Figure 10.1 on the previous page. Node I is the parent of J and O. Node G is a descendant of B. The search path to L is $\langle A, I, J, K, L\rangle$. The depth of N is 4. Node M is the node of maximum depth, and hence the height of the tree is 5. The height of the subtree rooted at B is 3. The height of the subtree rooted at H is 0. Nodes $D, E, H, M, N,$ and P are the leaves of the tree.

A *full binary tree* is a binary tree in which every node other than the leaves has two children. A *perfect binary tree* is a full binary tree in which all leaves are at the same depth, and in which every parent has two children. A *complete binary tree* is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. (This terminology is not universal, e.g., some authors use complete binary tree where we write perfect binary tree.) It is straightforward to prove using induction that the number of nonleaf nodes in a full binary tree is one less than the number of leaves. A perfect binary tree of height h contains exactly $2^{h+1} - 1$ nodes, of which 2^h are leaves. A complete binary tree on n nodes has height $\lfloor \lg n \rfloor$. A left-skewed tree is a tree in which no node has a right child; a right-skewed tree is a tree in which no node has a left child. In either case, we refer to the binary tree as being skewed.

A key computation on a binary tree is *traversing* all the nodes in the tree. (Traversing is also sometimes called *walking*.) Here are some ways in which this visit can be done.

- Traverse the left subtree, visit the root, then traverse the right subtree (an *inorder* traversal). An inorder traversal of the binary tree in Figure 10.1 on the preceding page visits the nodes in the following order: $\langle D, C, E, B, F, H, G, A, J, L, M, K, N, I, O, P\rangle$.
- Visit the root, traverse the left subtree, then traverse the right subtree (a *preorder* traversal). A preorder traversal of the binary tree in Figure 10.1 on the previous page visits the nodes in the following order: $\langle A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\rangle$.
- Traverse the left subtree, traverse the right subtree, and then visit the root (a *postorder* traversal). A postorder traversal of the binary tree in Figure 10.1 on the preceding page visits the nodes in the following order: $\langle D, E, C, H, G, F, B, M, L, N, K, J, P, O, I, A\rangle$.

Let T be a binary tree of n nodes, with height h. Implemented recursively, these traversals have $O(n)$ time complexity and $O(h)$ additional space complexity. (The space complexity is dictated by the maximum depth of the function call stack.) If each node has a parent field, the traversals can be done with $O(1)$ additional space complexity.

The term tree is overloaded, which can lead to confusion; see Page 316 for an overview of the common variants.

Binary trees boot camp

A good way to get up to speed with binary trees is to implement the three basic traversals—inorder, preorder, and postorder.

```
void TreeTraversal(const unique_ptr<BinaryTreeNode<int>>& root) {
  if (root) {
    // Preorder: Processes the root before the traversals of left and right
    // children.
    cout << "Preorder: " << root->data << endl;
    TreeTraversal(root->left);
    // Inorder: Processes the root after the traversal of left child and before
    // the traversal of right child.
    cout << "Inorder: " << root->data << endl;
    TreeTraversal(root->right);
    // Postorder: Processes the root after the traversals of left and right
    // children.
    cout << "Postorder: " << root->data << endl;
  }
}
```

The time complexity of each approach is $O(n)$, where n is the number of nodes in the tree. Although no memory is explicitly allocated, the function call stack reaches a maximum depth of h, the height of the tree. Therefore, the space complexity is $O(h)$. The minimum value for h is $\lg n$ (complete binary tree) and the maximum value for h is n (skewed tree).

Recursive algorithms are well-suited to problems on trees. Remember to include space implicitly allocated on the **function call stack** when doing space complexity analysis. [Problem 10.1]

Some tree problems have simple brute-force solutions that use $O(n)$ space solution, but subtler solutions that uses the **existing tree nodes** to reduce space complexity to $O(1)$. [Problem 10.14]

Consider **left- and right-skewed trees** when doing complexity analysis. Note that $O(h)$ complexity, where h is the tree height, translates into $O(\log n)$ complexity for balanced trees, but $O(n)$ complexity for skewed trees. [Problem 10.12]

If each node has a **parent field**, use it to make your code simpler, and to reduce time and space complexity. [Problem 10.10]

It's easy to make the **mistake** of treating a node that has a `single child` as a leaf. [Problem 10.6]

10.1 Test if a binary tree is height-balanced

A binary tree is said to be height-balanced if for each node in the tree, the difference in the height of its left and right subtrees is at most one. A perfect binary tree is height-balanced, as is a complete binary tree. A height-balanced binary tree does not have to be perfect or complete—see Figure 10.2 on the following page for an example.

Write a program that takes as input the root of a binary tree and checks whether the tree is height-balanced.

Hint: Think of a classic binary tree algorithm.

Solution: Here is a brute-force algorithm. Compute the height for the tree rooted at each node x recursively. The basic computation is to compute the height for each node starting from the leaves, and proceeding upwards. For each node, we check if the difference in heights of the left and right

Chapter

11 Heaps

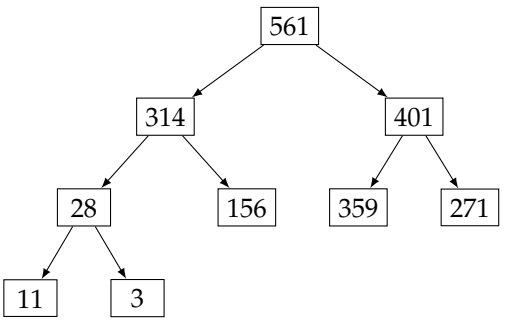
Using F-heaps we are able to obtain improved running times for several network optimization algorithms.

— "Fibonacci heaps and their uses,"
M. L. Fredman and R. E. Tarjan, 1987

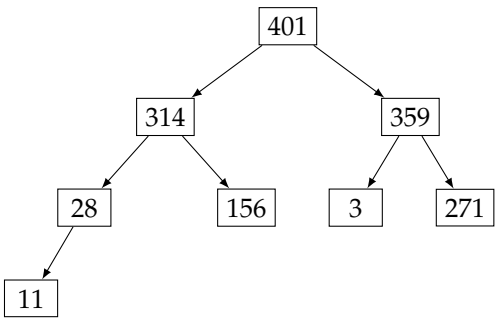
A *heap* (also referred to as a priority queue) is a specialized binary tree. Specifically, it is a complete binary tree as defined on Page 136. The keys must satisfy the *heap property*—the key at each node is at least as great as the keys stored at its children. See Figure 11.1(a) for an example of a max-heap. A max-heap can be implemented as an array; the children of the node at index i are at indices $2i + 1$ and $2i + 2$. The array representation for the max-heap in Figure 11.1(a) is $\langle 561, 314, 401, 28, 156, 359, 271, 11, 3\rangle$.

A max-heap supports $O(\log n)$ insertions, $O(1)$ time lookup for the max element, and $O(\log n)$ deletion of the max element. The extract-max operation is defined to delete and return the maximum element. See Figure 11.1(b) for an example of deletion of the max element. Searching for arbitrary keys has $O(n)$ time complexity.

The *min-heap* is a completely symmetric version of the data structure and supports $O(1)$ time lookups for the minimum element.



(a) A max-heap. Note that the root hold the maximum key, 561.



(b) After the deletion of max of the heap in (a). Deletion is performed by replacing the root's key with the key at the last leaf and then recovering the heap property by repeatedly exchanging keys with children.

Figure 11.1: A max-heap and deletion on that max-heap.

Heaps boot camp

Suppose you were asked to write a program which takes a sequence of strings presented in "streaming" fashion: you cannot back up to read an earlier value. Your program must compute the k longest

strings in the sequence. All that is required is the k longest strings—it is not required to order these strings.

As we process the input, we want to track the k longest strings seen so far. Out of these k strings, the string to be evicted when a longer string is to be added is the shortest one. A min-heap (not a max-heap!) is the right data structure for this application, since it supports efficient find-min, remove-min, and insert. In the program below we use a heap with a custom compare function, wherein strings are ordered by length.

```
vector<string> TopK(int k, istringstream* stream) {
  priority_queue<string, vector<string>, function<bool(string, string)>>
  min_heap([](const string& a, const string& b) -> bool {
    return a.size() >= b.size();
  });
  string next_string;
  while (*stream >> next_string) {
    min_heap.emplace(next_string);
    if (min_heap.size() > k) {
      // Remove the shortest string. Note that the comparison function above
      // will order the strings by length.
      min_heap.pop();
    }
  }
  vector<string> result;
  while (!min_heap.empty()) {
    result.emplace_back(min_heap.top());
    min_heap.pop();
  }
  return result;
}
```

Each string is processed in $O(\log k)$ time, which is the time to add and to remove the minimum element from the heap. Therefore, if there are n strings in the input, the time complexity to process all of them is $O(n \log k)$.

We could improve best-case time complexity by first comparing the new string's length with the length of the string at the top of the heap (getting this string takes $O(1)$ time) and skipping the insert if the new string is too short to be in the set.

> Use a heap when **all you care about** is the **largest** or **smallest** elements, and you **do not need** to support fast lookup, delete, or search operations for arbitrary elements. [Problem 11.1]
>
> A heap is a good choice when you need to compute the k **largest or** k **smallest** elements in a collection. For the former, use a min-heap, for the latter, use a max-heap. [Problem 11.4]

Know your heap libraries

The implementation of heaps in C++ is referred to as a priority queue; the class is `priority_queue`. The key functions are `push("Gauss")` (or `emplace("Gauss")`), `top()`, and `pop()`. Calling `top()` and `pop()` on an empty stack throws an exception. It is possible to specify a custom comparator in the heap constructor, as illustrated on on the current page.

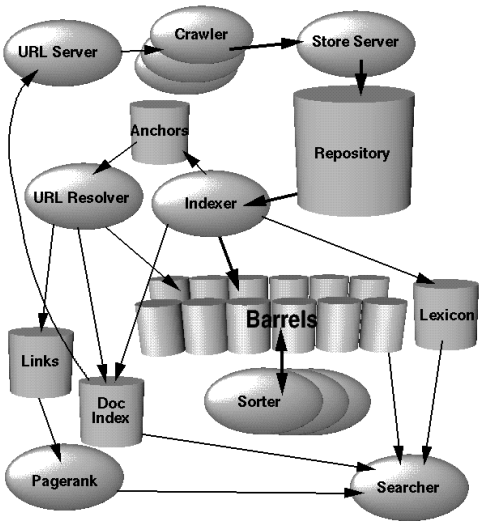
11.1 Merge sorted files

This problem is motivated by the following scenario. You are given 500 files, each containing stock trade information for an S&P 500 company. Each trade is encoded by a line in the following format:

Chapter

12

Searching



— *"The Anatomy of A Large-Scale Hypertextual Web Search Engine,"*
S. M. Brin and L. Page, 1998

Search algorithms can be classified in a number of ways. Is the underlying collection static or dynamic, i.e., inserts and deletes are interleaved with searching? Is it worth spending the computational cost to preprocess the data so as to speed up subsequent queries? Are there statistical properties of the data that can be exploited? Should we operate directly on the data or transform it?

In this chapter, our focus is on static data stored in sorted order in an array. Data structures appropriate for dynamic updates are the subject of Chapters 11, 13, and 15.

The first collection of problems in this chapter are related to binary search. The second collection pertains to general search.

Binary search

Given an arbitrary collection of n keys, the only way to determine if a search key is present is by examining each element. This has $O(n)$ time complexity. Fundamentally, binary search is a natural elimination-based strategy for searching a sorted array. The idea is to eliminate half the keys from consideration by keeping the keys in sorted order. If the search key is not equal to the middle element of the array, one of the two sets of keys to the left and to the right of the middle element can be eliminated from further consideration.

Questions based on binary search are ideal from the interviewers perspective: it is a basic technique that every reasonable candidate is supposed to know and it can be implemented in a few lines of code. On the other hand, binary search is much trickier to implement correctly than it appears—you should implement it as well as write corner case tests to ensure you understand it properly.

Many published implementations are incorrect in subtle and not-so-subtle ways—a study reported that it is correctly implemented in only five out of twenty textbooks. Jon Bentley, in his book "*Programming Pearls*" reported that he assigned binary search in a course for professional programmers and found that 90% failed to code it correctly despite having ample time. (Bentley's students would have been gratified to know that his own published implementation of binary search, in a column titled "Writing Correct Programs", contained a bug that remained undetected for over twenty years.)

Binary search can be written in many ways—recursive, iterative, different idioms for conditionals, etc. Here is an iterative implementation adapted from Bentley's book, which includes his bug.

```
int bsearch(int t, const vector<int>& A) {
  int L = 0, U = A.size() - 1;
  while (L <= U) {
    int M = (L + U) / 2;
    if (A[M] < t) {
      L = M + 1;
    } else if (A[M] == t) {
      return M;
    } else {
      U = M - 1;
    }
  }
  return -1;
}
```

The error is in the assignment `M = (L + U) / 2` in Line 4, which can potentially lead to overflow. This overflow can be avoided by using `M = L + (U - L) / 2`.

The time complexity of binary search is given by $T(n) = T(n/2) + c$, where c is a constant. This solves to $T(n) = O(\log n)$, which is far superior to the $O(n)$ approach needed when the keys are unsorted. A disadvantage of binary search is that it requires a sorted array and sorting an array takes $O(n \log n)$ time. However, if there are many searches to perform, the time taken to sort is not an issue.

Many variants of searching a sorted array require a little more thinking and create opportunities for missing corner cases.

Searching boot camp

When objects are comparable, they can be sorted and searched for using library search functions. Typically, the language knows how to compare built-in types, e.g., integers, strings, library classes for date, URLs, SQL timestamps, etc. However, user-defined types used in sorted collections must explicitly implement comparision, and ensure this comparision has basic properties such as transitivity. (If the comparision is implemented incorrectly, you may find a lookup into a sorted collection fails, even when the item is present.)

Suppose we are given as input an array of students, sorted by descending GPA, with ties broken on name. In the program below, we show how to use the library binary search routine to perform

fast searches in this array. In particular, we pass binary search a custom comparator which compares students on GPA (higher GPA comes first), with ties broken on name.

```
struct Student {
  string name;
  double grade_point_average;
};

const static function<bool(const Student&, const Student&)> CompGPA = [](
    const Student& a, const Student& b) -> bool {
  if (a.grade_point_average != b.grade_point_average) {
    return a.grade_point_average > b.grade_point_average;
  }
  return a.name < b.name;
};

bool SearchStudent(
    const vector<Student>& students, const Student& target,
    const function<bool(const Student&, const Student&)>& comp_GPA) {
  return binary_search(students.begin(), students.end(), target, comp_GPA);
}
```

Assuming the i-th element in the sequence can be accessed in $O(1)$ time, the time complexity of the program is $O(\log n)$.

Binary search is an effective search tool. It is applicable to more than just searching in **sorted arrays**, e.g., it can be used to search an **interval of real numbers or integers**. [Problem 12.4]

If your solution uses sorting, and the computation performed after sorting is faster than sorting, e.g., $O(n)$ or $O(\log n)$, **look for solutions that do not perform a complete sort**. [Problem 12.8]

Consider **time/space tradeoffs**, such as making multiple passes through the data. [Problem 12.9]

Know your searching libraries

Searching is a very broad concept, and it is present in many data structures. For example `find(A.begin(), A.end(), target)` in `algorithm` header finds the first element in a STL container. Here we focus on binary search in a sorted STL container.

- To check a targeted value is presented, use `binary_search(A.begin(), A.end(), target)`. Note that it returns a boolean about the status of existence instead of the location.
- To find the first element that is not less than a targeted value, use `lower_bound(A.begin(), A.end(), target)`. In other words, it find the first element that is greater than or equal to the targeted value.
- To find the first element that is greater than a targeted value, use `upper_bound(A.begin(), A.end(), target)`.

All three functions above have a time complexity of $O(\log n)$ in a sorted STL container containing n elements. In an interview, if it is allowed, use the above functions, instead of implementing your own binary search.

12.1 Search a sorted array for first occurrence of k

Binary search commonly asks for the index of *any* element of a sorted array that is equal to a specified element. The following problem has a slight twist on this.

Chapter

13 Hash Tables

The new methods are intended to reduce the amount of space required to contain the hash-coded information from that associated with conventional methods. The reduction in space is accomplished by exploiting the possibility that a small fraction of errors of commission may be tolerable in some applications.

— "*Space/time trade-offs in hash coding with allowable errors*,"
B. H. Bloom, 1970

The idea underlying a *hash table* is to store objects according to their key field in an array. Objects are stored in array locations ("slots") based on the "hash code" of the key. The hash code is an integer computed from the key by a hash function. If the hash function is chosen well, the objects are distributed uniformly across the array locations.

If two keys map to the same location, a "collision" is said to occur. The standard mechanism to deal with collisions is to maintain a linked list of objects at each array location. If the hash function does a good job of spreading objects across the underlying array and take $O(1)$ time to compute, on average, lookups, insertions, and deletions have $O(1 + n/m)$ time complexity, where n is the number of objects and m is the length of the array. If the "load" n/m grows large, rehashing can be applied to the hash table. A new array with a larger number of locations is allocated, and the objects are moved to the new array. Rehashing is expensive ($O(n + m)$ time) but if it is done infrequently (for example, whenever the number of entries doubles), its amortized cost is low.

A hash table is qualitatively different from a sorted array—keys do not have to appear in order, and randomization (specifically, the hash function) plays a central role. Compared to binary search trees (discussed in Chapter 15), inserting and deleting in a hash table is more efficient (assuming rehashing is infrequent). One disadvantage of hash tables is the need for a good hash function but this is rarely an issue in practice. Similarly, rehashing is not a problem outside of realtime systems and even for such systems, a separate thread can do the rehashing.

A hash function has one hard requirement—equal keys should have equal hash codes. This may seem obvious, but is easy to get wrong, e.g., by writing a hash function that is based on address rather than contents, or by including profiling data.

A softer requirement is that the hash function should "spread" keys, i.e., the hash codes for a subset of objects should be uniformly distributed across the underlying array. In addition, a hash function should be efficient to compute.

A common mistake with hash tables is that a key that's present in a hash table will be updated. The consequence is that a lookup for that key will now fail, even though it's still in the hash table. If you have to update a key, first remove it, then update it, and finally, add it back—this ensures it's moved the the correct array location. As a rule, you should avoid using mutable objects as keys.

Now we illustrate the steps in designing a hash function suitable for strings. First, the hash function should examine all the characters in the string. It should give a large range of values, and

should not let one character dominate (e.g., if we simply cast characters to integers and multiplied them, a single 0 would result in a hash code of 0). We would also like a rolling hash function, one in which if a character is deleted from the front of the string, and another added to the end, the new hash code can be computed in $O(1)$ time (see Solution 7.13 on Page 98). The following function has these properties:

```
int StringHash(const string& str, int modulus) {
  const int kMult = 997;
  int val = 0;
  for (char c : str) {
    val = (val * kMult + c) % modulus;
  }
  return val;
}
```

A hash table is a good data structure to represent a dictionary, i.e., a set of strings. In some applications, a trie, which is a tree data structure that is used to store a dynamic set of strings, has computational advantages. Unlike a BST, nodes in the tree do not store a key. Instead, the node's position in the tree defines the key which it is associated with. See Problem 25.20 on Page 419 for an example of trie construction and application.

Hash tables boot camp

We introduce hash tables with two examples—one is an application that benefits from the algorithmic advantages of hash tables, and the other illustrates the design of a class that can be used in a hash table.

An application of hash tables

Anagrams are popular word play puzzles, whereby rearranging letters of one set of words, you get another set of words. For example, "eleven plus two" is an anagram for "twelve plus one". Crossword puzzle enthusiasts and Scrabble players benefit from the ability to view all possible anagrams of a given set of letters.

Suppose you were asked to write a program that takes as input a set of words and returns groups of anagrams for those words. Each group must contain at least two words.

For example, if the input is "debitcard", "elvis", "silent", "badcredit", "lives", "freedom", "listen", "levis", "money" then there are three groups of anagrams: (1.) "debitcard", "badcredit"; (2.) "elvis", "lives", "levis"; (3.) "silent", "listen". (Note that "money" does not appear in any group, since it has no anagrams in the set.)

Let's begin by considering the problem of testing whether one word is an anagram of another. Since anagrams do not depend on the ordering of characters in the strings, we can perform the test by sorting the characters in the string. Two words are anagrams if and only if they result in equal strings after sorting. For example, sort("logarithmic") and sort("algorithmic") are both "acghiilmort", so "logarithmic" and "algorithmic" are anagrams.

We can form the described grouping of strings by iterating through all strings, and comparing each string with all other remaining strings. If two strings are anagrams, we do not consider the second string again. This leads to an $O(n^2 m \log m)$ algorithm, where n is the number of strings and m is the maximum string length.

Looking more carefully at the above computation, note that the key idea is to map strings to a representative. Given any string, its sorted version can be used as a unique identifier for the

```
  vector<string> names;
};

// Hash function for ContactList.
struct HashContactList {
  size_t operator()(const ContactList& contacts) const {
    size_t hash_code = 0;
    for (const string& name :
         unordered_set<string>(contacts.names.begin(), contacts.names.end())) {
      hash_code ^= hash<string>()(name);
    }
    return hash_code;
  }
};

vector<ContactList> MergeContactLists(const vector<ContactList>& contacts) {
  unordered_set<ContactList, HashContactList> unique_contacts(contacts.begin(),
                                                              contacts.end());
  return {unique_contacts.begin(), unique_contacts.end()};
}
```

The time complexity of computing the hash is $O(n)$, where n is the number of strings in the contact list. Hash codes are often cached for performance, with the caveat that the cache must be cleared if object fields that are referenced by the hash function are updated.

Hash tables have the **best theoretical and real-world performance** for lookup, insert and delete. Each of these operations has $O(1)$ time complexity. The $O(1)$ time complexity for insertion is for the average case—a single insert can take $O(n)$ if the hash table has to be resized. [Problem 13.2]

Consider using a hash code as a **signature** to enhance performance, e.g., to filter out candidates. [Problem 13.14]

Consider using a precomputed `lookup table` instead of boilerplate if-then code for mappings, e.g., from character to value, or character to character. [Problem 7.9]

When defining your own type that will be put in a hash table, be sure you understand the relationship between **logical equality** and the fields the hash function must inspect. Specifically, anytime equality is implemented, it is imperative that the correct hash function is also implemented, otherwise when objects are placed in hash tables, logically equivalent objects may appear in different buckets, leading to lookups returning false, even when the searched item is present.

Sometimes you'll need a `multimap`, i.e., a map that contains multiple values for a single key, or a bi-directional map. If the language's standard libraries do not provide the functionality you need, learn how to implement a multimap using lists as values, or find a **third party library** that has a multimap.

Know your hash table libraries

There are two hash table-based data structures commonly used in C++—`unordered_set` and `unordered_map`. The difference between the two is that the latter stores key-value pairs, whereas the former simply stores keys. Both have the property that they do not allow for duplicate keys, unlike, for example, `list` and `priority_queue`.

The most important functions for `unordered_set` are `insert(42)` (or `emplace(42)`), `erase(42)`, `find(42)`, and `size()`.

- `insert(val)` inserts new element and returns a pair of iterator and boolean where the iterator points to the newly inserted element or the element whose key is equivalent, and the boolean indicating if the element was added successfully, i.e., was not already present.
- `find(k)` returns the iterator to the element if it was present; otherwise, a special iterator `end()` is returned.
- The order in which keys are traversed by the iterator returned by `begin()` is unspecified; it may even change with time.

The most important methods for `unordered_map` are `insert({42, "Gauss"})` (or `emplace({42, "Gauss"})`), `erase(42)`, `find(42)`, and `size()`. Those functions are analogous to the ones in `unordered_set`. The `pair<key, value>` type is a key-value pair that's useful when iterating over the map. The iteration order is not fixed, though iterations over the entry set, the key set, and the value set do agree.

The `hash()` method in `functional` header provides convenient functions for hashing the basic classes in C++, e.g., `int`, `bool`, `string`, `unique_ptr`, `shared_ptr`, etc. We show how to design a hash function for a custom class on Page 189.

13.1 Test for palindromic permutations

A palindrome is a string that reads the same forwards and backwards, e.g., "level", "rotator", and "foobaraboof".

Write a program to test whether the letters forming a string can be permuted to form a palindrome. For example, "edified" can be permuted to form "deified".

Hint: Find a simple characterization of strings that can be permuted to form a palindrome.

Solution: A brute-force approach is to compute all permutations of the string, and test each one for palindromicity. This has a very high time complexity. Examining the approach in more detail, one thing to note is that if a string begins with say 'a', then we only need consider permutations that end with 'a'. This observation can be used to prune the permutation-based algorithm. However, a more powerful conclusion is that all characters must occur in pairs for a string to be permutable into a palindrome, with one exception, if the string is of odd length. For example, for the string "edified", which is of odd length (7) there are two 'e', two 'f's, two 'i's, and one 'd'—this is enough to guarantee that "edified" can be permuted into a palindrome.

More formally, if the string is of even length, a necessary and sufficient condition for it to be a palindrome is that each character in the string appear an even number of times. If the length is odd, all but one character should appear an even number of times. Both these cases are covered by testing that at most one character appears an odd number of times, which can be checked using a hash table mapping characters to frequencies.

```
bool CanFormPalindrome(const string& s) {
  unordered_map<char, int> char_frequencies;
  // Compute the frequency of each char in s.
  for (char c : s) {
    ++char_frequencies[c];
  }
```

Chapter

14 Sorting

PROBLEM 14 *(Meshing). Two monotone sequences S, T, of lengths n, m, respectively, are stored in two systems of $n(p+1), m(p+1)$ consecutive memory locations, respectively: $s, s+1, \ldots, s+n(p+1)-1$ and $t, t+1, \ldots, t+m(p+1)-1$. ... It is desired to find a monotone permutation R of the sum $[S, T]$, and place it at the locations $r, r+1, \ldots, r+(n+m)(p+1)-1$.*

— "*Planning And Coding Of Problems For An Electronic Computing Instrument*," H. H. Goldstine and J. von Neumann, 1948

Sorting—rearranging a collection of items into increasing or decreasing order—is a common problem in computing. Sorting is used to preprocess the collection to make searching faster (as we saw with binary search through an array), as well as identify items that are similar (e.g., students are sorted on test scores).

Naive sorting algorithms run in $O(n^2)$ time. A number of sorting algorithms run in $O(n \log n)$ time—heapsort, merge sort, and quicksort are examples. Each has its advantages and disadvantages: for example, heapsort is in-place but not stable; merge sort is stable but not in-place; quicksort runs $O(n^2)$ time in worst-case. (An in-place sort is one which uses $O(1)$ space; a stable sort is one where entries which are equal appear in their original order.)

A well-implemented quicksort is usually the best choice for sorting. We briefly outline alternatives that are better in specific circumstances.

For short arrays, e.g., 10 or fewer elements, insertion sort is easier to code and faster than asymptotically superior sorting algorithms. If every element is known to be at most k places from its final location, a min-heap can be used to get an $O(n \log k)$ algorithm (Solution 11.3 on Page 162). If there are a small number of distinct keys, e.g., integers in the range [0..255], counting sort, which records for each element, the number of elements less than it, works well. This count can be kept in an array (if the largest number is comparable in value to the size of the set being sorted) or a BST, where the keys are the numbers and the values are their frequencies. If there are many duplicate keys we can add the keys to a BST, with linked lists for elements which have the same key; the sorted result can be derived from an in-order traversal of the BST.

Most sorting algorithms are not stable. Merge sort, carefully implemented, can be made stable. Another solution is to add the index as an integer rank to the keys to break ties.

Most sorting routines are based on a compare function that takes two items as input and returns -1 if the first item is smaller than the second item, 0 if they are equal and 1 otherwise. However, it is also possible to use numerical attributes directly, e.g., in radix sort.

The heap data structure is discussed in detail in Chapter 11. Briefly, a max-heap (min-heap) stores keys drawn from an ordered set. It supports $O(\log n)$ inserts and $O(1)$ time lookup for the maximum (minimum) element; the maximum (minimum) key can be deleted in $O(\log n)$ time. Heaps can be helpful in sorting problems, as illustrated by Problems 11.1 on Page 159, 11.2 on Page 161, and 11.3 on Page 162.

Sorting boot camp

It's important to know how to use effectively use the sort functionality provided by your language's library. Let's say we are given a student class that implements a compare method that compares students by name. To sort an array of students by GPA, we pass an explicit compare function to the sort function.

```
struct Student {
  bool operator<(const Student& that) const { return name < that.name; }

  string name;
  double grade_point_average;
};

void SortByGPA(vector<Student>* students) {
  sort(students->begin(), students->end(),
       [](const Student& a, const Student& b) -> bool {
         return a.grade_point_average >= b.grade_point_average;
       });
}

void SortByName(vector<Student>* students) {
  // Rely on the operator< defined in Student.
  sort(students->begin(), students->end());
}
```

The time complexity of any reasonable library sort is $O(n \log n)$ for an array with n entries. Most library sorting functions are based on quicksort, which has $O(1)$ space complexity.

Sorting problems come in two flavors: (1.) **use sorting to make subsequent steps in an algorithm simpler**, and (2.) design a **custom sorting routine**. For the former, it's fine to use a library sort function, possibly with a custom comparator. For the latter, use a data structure like a BST, heap, or array indexed by values. [Problems 14.4 and 14.7]

The most natural reason to sort is if the inputs have a **natural ordering**, and sorting can be used as a preprocessing step to **speed up searching**. [Problem 14.5]

For `specialized input`, e.g., a very small range of values, or a small number of values, it's possible to sort in $O(n)$ time rather than $O(n \log n)$ time. [Problems 6.1 and 14.7]

It's often the case that sorting can be implemented in **less space** than required by a brute-force approach. [Problem 14.2]

Know your sorting libraries

To sort an array, use `sort()` in the `algorithm` header, and to sort a list use the member function `list::sort()`.

- The time complexity of sorting is $O(n \log n)$, where n is length of the array. The standard gives no guarantees as to the space complexity. In practice, it's most commonly a variant of quicksort, which does not allocate additional memory, but uses $O(\log n)$ space on the function call stack.
- Both `sort()` in `algorithm` and `list::sort()` operate on arrays and lists of objects that implement `operator<()`.

- Both `sort()` in `algorithm` and `list::sort()` have the provision of sorting according to an explicit comparator object, as shown on on the preceding page.

14.1 Compute the intersection of two sorted arrays

A natural implementation for a search engine is to retrieve documents that match the set of words in a query by maintaining an inverted index. Each page is assigned an integer identifier, its *document-ID*. An inverted index is a mapping that takes a word w and returns a sorted array of page-ids which contain w—the sort order could be, for example, the page rank in descending order. When a query contains multiple words, the search engine finds the sorted array for each word and then computes the intersection of these arrays—these are the pages containing all the words in the query. The most computationally intensive step of doing this is finding the intersection of the sorted arrays.

Write a program which takes as input two sorted arrays, and returns a new array containing elements that are present in both of the input arrays. The input arrays may have duplicate entries, but the returned array should be free of duplicates. For example, the input is $\langle 2, 3, 3, 5, 5, 6, 7, 7, 8, 12\rangle$ and $\langle 5, 5, 6, 8, 8, 9, 10, 10\rangle$, your output should be $\langle 5, 6, 8\rangle$.

Hint: Solve the problem if the input array lengths differ by orders of magnitude. What if they are approximately equal?

Solution: The brute-force algorithm is a "loop join", i.e., traversing through all the elements of one array and comparing them to the elements of the other array. Let m and n be the lengths of the two input arrays.

```
vector<int> IntersectTwoSortedArrays(const vector<int>& A,
                                     const vector<int>& B) {
  vector<int> insersection_A_B;
  for (int i = 0; i < A.size(); ++i) {
    if (i == 0 || A[i] != A[i - 1]) {
      for (int b : B) {
        if (A[i] == b) {
          insersection_A_B.emplace_back(A[i]);
          break;
        }
      }
    }
  }
  return insersection_A_B;
}
```

The brute-force algorithm has $O(mn)$ time complexity.

Since both the arrays are sorted, we can make some optimizations. First, we can iterate through the first array and use binary search in array to test if the element is present in the second array.

```
vector<int> IntersectTwoSortedArrays(const vector<int>& A,
                                     const vector<int>& B) {
  vector<int> intersection_A_B;
  for (int i = 0; i < A.size(); ++i) {
    if ((i == 0 || A[i] != A[i - 1]) &&
        binary_search(B.cbegin(), B.cend(), A[i])) {
      intersection_A_B.emplace_back(A[i]);
    }
  }
  return intersection_A_B;
```

Chapter

15 Binary Search Trees

The number of trees which can be formed with $n+1$ given knots $\alpha, \beta, \gamma, \ldots = (n+1)^{n-1}$.

— "*A Theorem on Trees*," A. Cayley, 1889

Adding and deleting elements to an array is computationally expensive, when the array needs to stay sorted. BSTs are similar to arrays in that the stored values (the "keys") are stored in a sorted order. BSTs offer the ability to search for a key as well as find the *min* and *max* elements, look for the successor or predecessor of a search key (which itself need not be present in the BST), and enumerate the keys in a range in sorted order. However, unlike with a sorted array, keys can be added to and deleted from a BST efficiently.

A BST is a binary tree as defined in Chapter 10 in which the nodes store keys that are comparable, e.g., integers or strings. The keys stored at nodes have to respect the BST property—the key stored at a node is greater than or equal to the keys stored at the nodes of its left subtree and less than or equal to the keys stored in the nodes of its right subtree. Figure 15.1 on the next page shows a BST whose keys are the first 16 prime numbers.

Key lookup, insertion, and deletion take time proportional to the height of the tree, which can in worst-case be $O(n)$, if insertions and deletions are naively implemented. However, there are implementations of insert and delete which guarantee that the tree has height $O(\log n)$. These require storing and updating additional data at the tree nodes. Red-black trees are an example of height-balanced BSTs and are widely used in data structure libraries.

A common mistake with BSTs is that an object that's present in a BST is be updated. The consequence is that a lookup for that object will now fail, even though it's still in the BST. When a mutable object that's in a BST is to be updated, first remove it from the tree, then update it, then add it back. (As a rule, avoid putting mutable objects in a BST.)

The BST prototype is as follows:

```
template <typename T>
struct BSTNode {
  T data;
  unique_ptr<BSTNode<T>> left, right;
};
```

Binary search trees boot camp

Searching is the single most fundamental application of BSTs. Unlike a hash table, a BST offers the ability to find the min and max elements, and find the next largest/next smallest element. These operations, along with lookup, delete and find, take time $O(\log n)$ for library implementations of BSTs. Both BSTs and hash tables use $O(n)$ space—in practice, a BST uses slightly more space.

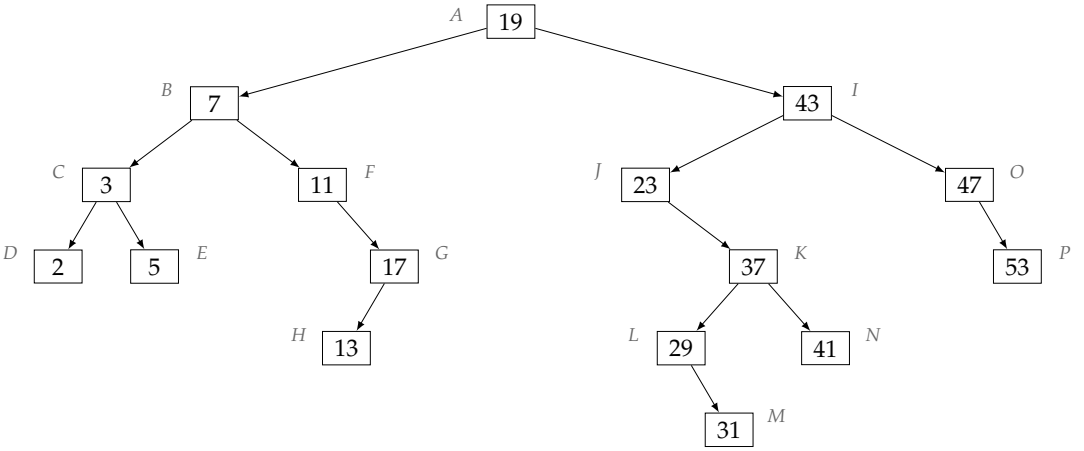


Figure 15.1: An example of a BST.

The following program demonstrates how to check if a given value is present in a BST. It is a nice illustration of the power of recursion when operating on BSTs.

```
BSTNode<int>* SearchBST(const unique_ptr<BSTNode<int>>& tree, int key) {
  if (tree == nullptr) {
    return nullptr;
  }
  if (tree->data == key) {
    return tree.get();
  }
  return key < tree->data ? SearchBST(tree->left, key)
                          : SearchBST(tree->right, key);
}
```

Since the program descends tree with in each step, and spends $O(1)$ time per level, the time complexity is $O(h)$, where h is the height of the tree.

With a BST you can **iterate** through elements in **sorted order** in time $O(n)$ (regardless of whether it is balanced). [Problem 15.2]

Some problems need a **combination of a BST and a hashtable**. For example, if you insert student objects into a BST and entries are ordered by GPA, and then a student's GPA needs to be updated and all we have is the student's name and new GPA, we cannot find the student by name without a full traversal. However, with an additional hash table, we can directly go to the corresponding entry in the tree. [Problem 15.8]

Sometimes, it's necessary to **augment** a BST, e.g., the number of nodes at a subtree in addition to its key, or the range of values sorted in the subtree. [Problem 15.13]

The BST property is a **global property**—a binary tree may have the property that each node's key is greater than the key at its left child and smaller than the key at its right child, but it may not be a BST. [Problem 15.1]

Know your binary search tree libraries

There are two BST-based data structures commonly used in C++—`set` and `map`. Similar to `unordered_set` and `unordered_map` on Page 190, `set` stores keys, and `map` stores key-value pairs.

Below, we describe the functionalities added by `set` that go beyond what's in `unordered_set`. The functionalities added by `map` are similar.

- The iterator returned by `begin()` traverses keys in ascending order. (To iterate over keys in descending order, use `rbegin()`.)
- `*begin()`/`*rbegin()` yield the smallest and largest keys in the BST.
- `lower_bound(12)`/`upper_bound(3)` return the first element that is greater than or equal to/-greater than the argument. For example, if the vector is ⟨1010102020203030⟩, then `lower_bound (v.begin(), v.end(), 20)` returns 3 and `upper_bound (v.begin(), v.end(), 20)` returns 6.
- `equal_range(10)` return the range of values equal to the argument.

It's particularly important to note that these operations take $O(\log n)$, since they are backed by the underlying tree.

The `set` and `map` constructors permit for an explicit comparator object that is used to order keys, and you should be comfortable with the syntax used to specify this object. (See on Page 212 for an example of comparator syntax.)

15.1 Test if a binary tree satisfies the BST property

Write a program that takes as input a binary tree and checks if the tree satisfies the BST property.

Hint: Is it correct to check for each node that its key is greater than or equal to the key at its left child and less than or equal to the key at its right child?

Solution: A direct approach, based on the definition of a BST, is to begin with the root, and compute the maximum key stored in the root's left subtree, and the minimum key in the root's right subtree. We check that the key at the root is greater than or equal to the maximum from the left subtree and less than or equal to the minimum from the right subtree. If both these checks pass, we recursively check the root's left and right subtrees. If either check fails, we return false.

Computing the minimum key in a binary tree is straightforward: we take the minimum of the key stored at its root, the minimum key of the left subtree, and the minimum key of the right subtree. The maximum key is computed similarly. Note that the minimum can be in either subtree, since a general binary tree may not satisfy the BST property.

The problem with this approach is that it will repeatedly traverse subtrees. In the worst-case, when the tree is a BST and each node's left child is empty, the complexity is $O(n^2)$, where n is the number of nodes. The complexity can be improved to $O(n)$ by caching the largest and smallest keys at each node; this requires $O(n)$ additional storage for the cache.

We now present two approaches which have $O(n)$ time complexity and $O(h)$ additional space complexity, where h is the height of the tree.

The first approach is to check constraints on the values for each subtree. The initial constraint comes from the root. Every node in its left (right) subtree must have a key less than or equal (greater than or equal) to the key at the root. This idea generalizes: if all nodes in a tree must have keys in the range $[l, u]$, and the key at the root is w (which itself must be between $[l, u]$, otherwise the requirement is violated at the root itself), then all keys in the left subtree must be in the range $[l, w]$, and all keys stored in the right subtree must be in the range $[w, u]$.

Chapter

16 Recursion

The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

— "*Algorithms + Data Structures = Programs*,"
N. E. Wirth, 1976

Recursion is a method where the solution to a problem depends partially on solutions to smaller instances of related problems. Two key ingredients to a successful use of recursion are identifying the base cases, which are to be solved directly, and ensuring progress, that is the recursion converges to the solution.

A divide-and-conquer algorithm works by repeatedly decomposing a problem into two or more smaller independent subproblems of the same kind, until it gets to instances that are simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. Merge sort and quicksort are classical examples of divide-and-conquer.

Divide-and-conquer is not synonymous with recursion. In divide-and-conquer, the problem is divided into two or more independent smaller problems that are of the same type as the original problem. Recursion is more general—there may be a single subproblem, e.g., binary search, the subproblems may not be independent, e.g., dynamic programming, and they may not be of the same type as the original, e.g., regular expression matching. In addition, sometimes to improve runtime, and occasionally to reduce space complexity, a divide-and-conquer algorithm is implemented using iteration instead of recursion.

Recursion boot camp

The Euclidean algorithm for calculating the greatest common divisor (GCD) of two numbers is a classic example of recursion. The central idea is that if $y > x$, the GCD of x and y is the GCD of x and $y - x$. For example, $\text{GCD}(156, 36) = \text{GCD}((156 - 36) = 120, 36)$. By extension, this implies that the GCD of x and y is the GCD of x and $y \bmod x$, i.e., $\text{GCD}(156, 36) = \text{GCD}((156 \bmod 36) = 12, 36) = \text{GCD}(12, 36 \bmod 12 = 0) = 12$.

```
long long GCD(long long x, long long y) { return y == 0 ? x : GCD(y, x % y); }
```

Since with each recursive step one of the arguments is at least halved, it means that the time complexity is $O(\log \max(x, y))$. Put another way, the time complexity is $O(n)$, where n is the number of bits needed to represent the inputs. The space complexity is also $O(n)$, which is the maximum depth of the function call stack. (The program is easily converted to one which loops, thereby reducing the space complexity to $O(1)$.)

Recursion is especially suitable when the **input is expressed using recursive rules**. [Problem 25.26]

Recursion is a good choice for **search**, **enumeration**, and **divide-and-conquer**. [Problems 16.2, 16.9, and 25.27]

Use recursion as **alternative to deeply nested iteration** loops. For example, recursion is much better when you have an undefined number of levels, such as the IP address problem generalized to k substrings. [Problem 7.10]

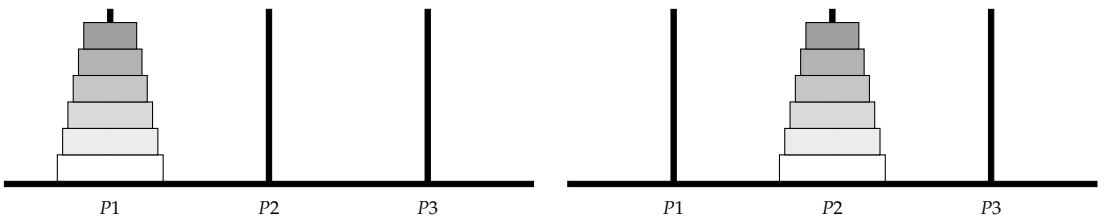
If you are asked to **remove recursion** from a program, consider mimicking call stack with the **stack data structure**. bt-traversal

Recursion can be easily removed from a **tail-recursive** program by using a while-loop—no stack is needed. (Optimizing compilers do this.) [Problem 5.7]

If a recursive function may end up being called with the **same arguments** more than once, **cache** the results—this is the idea behind Dynamic Programming (Chapter 17).

16.1 The Towers of Hanoi problem

A peg contains rings in sorted order, with the largest ring being the lowest. You are to transfer these rings to another peg, which is initially empty. This is illustrated in Figure 16.1.



(a) Initial configuration. (b) Desired configuration.

Figure 16.1: Tower of Hanoi with 6 pegs.

Write a program which prints a sequence of operations that transfers n rings from one peg to another. You have a third peg, which is initially empty. The only operation you can perform is taking a single ring from the top of one peg and placing it on the top of another peg. You must never place a larger ring above a smaller ring.

Hint: If you know how to transfer the top $n-1$ rings, how does that help move the nth ring?

Solution: The insight to solving this problem can be gained by trying examples. The three ring transfer can be achieved by moving the top two rings to the third peg, then moving the lowest ring (which is the largest) to the second peg, and then transferring the two rings on the third peg to the second peg, using the first peg as the intermediary. To transfer four rings, move the top three rings to the third peg, then moving the lowest ring (which is the largest) to the second peg, and then transfer the three rings on the third peg to the second peg, using the first peg as an intermediary. For both the three ring and four ring transfers, the first and third steps are instances of the same problem, which suggests the use of recursion. This approach is illustrated in Figure 16.2 on the facing page. Code implementing this idea is given below.

Chapter

17 Dynamic Programming

The important fact to observe is that we have attempted to solve a maximization problem involving a particular value of x and a particular value of N by first solving the general problem involving an arbitrary value of x and an arbitrary value of N.

— "Dynamic Programming,"
R. E. Bellman, 1957

DP is a general technique for solving optimization, search, and counting problems that can be decomposed into subproblems. You should consider using DP whenever you have to make choices to arrive at the solution, specifically, when the solution relates to subproblems.

Like divide-and-conquer, DP solves the problem by combining the solutions of multiple smaller problems, but what makes DP different is that the same subproblem may reoccur. Therefore, a key to making DP efficient is caching the results of intermediate computations. Problems whose solutions use DP are a popular choice for hard interview questions.

To illustrate the idea underlying DP, consider the problem of computing Fibonacci numbers. The first two Fibonacci numbers are 0 and 1. Successive numbers are the sums of the two previous numbers. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The Fibonacci numbers arise in many diverse applications—biology, data structure analysis, and parallel computing are some examples.

Mathematically, the nth Fibonacci number $F(n)$ is given by the equation $F(n) = F(n-1) + F(n-2)$, with $F(0) = 0$ and $F(1) = 1$. A function to compute $F(n)$ that recursively invokes itself has a time complexity that is exponential in n. This is because the recursive function computes some $F(i)$s repeatedly. Figure 17.1 on the next page graphically illustrates how the function is repeatedly called with the same arguments.

Caching intermediate results makes the time complexity for computing the nth Fibonacci number linear in n, albeit at the expense of $O(n)$ storage. The program below computes $F(n)$ via iteration in $O(n)$ time. Compared to a recursive program with caching, the iterative program fills in the cache in a bottom-up fashion, and reuses storage to reduce space complexity to $O(1)$.

```
unordered_map<int, int> cache;

int Fibonacci(int n) {
  if (n <= 1) {
    return n;
  } else if (!cache.count(n)) {
    cache[n] = Fibonacci(n - 1) + Fibonacci(n - 2);
  }
  return cache[n];
}
```

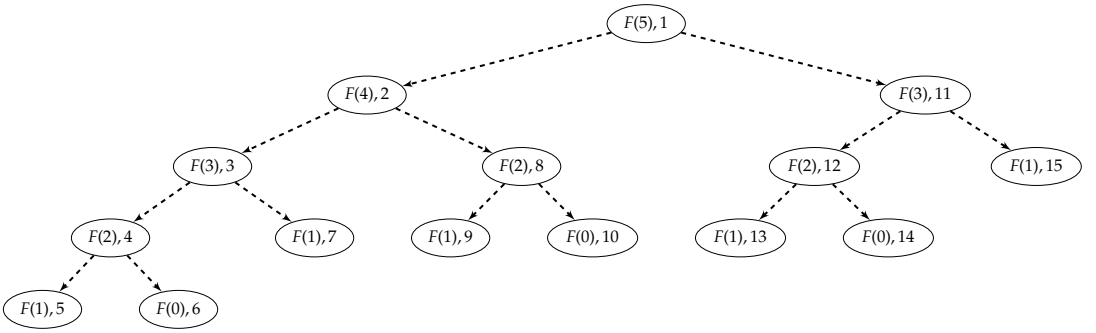


Figure 17.1: Tree of recursive calls when naively computing the 5th Fibonacci number, $F(5)$. Each node is a call: $F(x)$ indicates a call with argument x, and the italicized numbers on the right are the sequence in which calls take place. The children of a node are the subcalls made by that call. Note how there are 2 calls to $F(3)$, and 3 calls to each of $F(2)$, $F(1)$, and $F(0)$.

The key to solving a DP problem efficiently is finding a way to break the problem into subproblems such that

- the original problem can be solved relatively easily once solutions to the subproblems are available, and
- these subproblem solutions are cached.

Usually, but not always, the subproblems are easy to identify.

Here is a more sophisticated application of DP. Consider the following problem: find the maximum sum over all subarrays of a given array of integer. As a concrete example, the maximum subarray for the array in Figure 17.2 starts at index 0 and ends at index 3.

904	**40**	**523**	**12**	**-335**	**-385**	**-124**	**481**	**-31**
$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$

Figure 17.2: An array with a maximum subarray sum of 1479.

The brute-force algorithm, which computes each subarray sum, has $O(n^3)$ time complexity—there are $\frac{n(n+1)}{2}$ subarrays, and each subarray sum takes $O(n)$ time to compute. The brute-force algorithm can be improved to $O(n^2)$, at the cost of $O(n)$ additional storage, by first computing $S[k] = \sum A[0:k]$ for all k. The sum for $A[i:j]$ is then $S[j] - S[i-1]$, where $S[-1]$ is taken to be 0.

Here is a natural divide-and-conquer algorithm. Take $m = \lfloor \frac{n}{2} \rfloor$ to be the middle index of A. Solve the problem for the subarrays $L = A[0:m]$ and $R = A[m+1:n-1]$. In addition to the answers for each, we also return the maximum subarray sum l for a subarray ending at the last entry in L, and the maximum subarray sum r for a subarray starting at the first entry of R. The maximum subarray sum for A is the maximum of $l + r$, the answer for L, and the answer for R. The time complexity analysis is similar to that for quicksort, and the time complexity is $O(n \log n)$. Because of off-by-one errors, it takes some time to get the program just right.

Now we will solve this problem by using DP. A natural thought is to assume we have the solution for the subarray $A[0:n-2]$. However, even if we knew the largest sum subarray for subarray $A[0:n-2]$, it does not help us solve the problem for $A[0:n-1]$. Instead we need to know the subarray amongst all subarrays $A[0:i]$, $i < n-1$, with the smallest subarray sum. The desired value is $S[n-1]$ minus this subarray's sum. We compute this value by iterating through

the array. For each index j, the maximum subarray sum ending at j is equal to $S[j] - \min_{k \le j} S[k]$. During the iteration, we track the minimum $S[k]$ we have seen so far and compute the maximum subarray sum for each index. The time spent per index is constant, leading to an $O(n)$ time and $O(1)$ space solution. It is legal for all array entries to be negative, or the array to be empty. The algorithm handles these input cases correctly.

```
int FindMaximumSubarray(const vector<int>& A) {
  int min_sum = 0, sum = 0, max_sum = 0;
  for (int i = 0; i < A.size(); ++i) {
    sum += A[i];
    if (sum < min_sum) {
      min_sum = sum;
    }
    if (sum - min_sum > max_sum) {
      max_sum = sum - min_sum;
    }
  }
  return max_sum;
}
```

Here are some common mistakes made when applying DP.

- A common **mistake** in solving DP problems is trying to think of the recursive case by splitting the problem into **two equal halves**, *a la* quicksort, i.e., solve the subproblems for subarrays $A[0 : \lfloor \frac{n}{2} \rfloor]$ and $A[\lfloor \frac{n}{2} \rfloor + 1 : n]$ and combine the results. However, in most cases, these two subproblems are not sufficient to solve the original problem.
- Make sure that **combining solutions** to the subproblems does **yield a solution** to the original problem. For example, if the longest path without repeated cities from City 1 to City 2 passes through City 3, then the subpaths from City 1 to City 3 and City 3 to City 2 may not be individually longest paths without repeated cities.

Dynamic programming boot camp

The programs presented in the introduction for computing the Fibonacci numbers and the maximum subarray sum are good illustrations of DP.

Consider using DP whenever you have to **make choices** to arrive at the solution, specifically, when the solution relates to subproblems.

In addition to optimization problems, DP is also **applicable** to **counting and decision problems**—any problem where you can express a solution recursively in terms of the same computation on smaller instances.

Although conceptually DP involves recursion, often for efficiency the cache is **built "bottom-up"**, i.e., iteratively. [Problem 17.3].

To save space, `cache space` may be `recycled` once it is know that a set of entries will not be looked up again. [Problems 17.1 and 17.2]

Sometimes, `recursion may out-perform a bottom-up DP` solution, e.g., when the solution is found early or subproblems can be `pruned` through bounding. [Problem 17.5]

Chapter

18

Greedy Algorithms and Invariants

The intended function of a program, or part of a program, can be specified by making general assertions about the values which the relevant variables will take after *execution of the program.*

— "*An Axiomatic Basis for Computer Programming*,"
C. A. R. Hoare, 1969

Greedy algorithms

A greedy algorithm is an algorithm that computes a solution in steps; at each step the algorithm makes a decision that is locally optimum, and it never changes that decision.

The example on Page 35 illustrates how different greedy algorithms for the same problem can differ in terms of optimality. As another example, consider making change for 48 pence in the old British currency where the coins came in $30, 24, 12, 6, 3,$ and 1 pence denominations. Suppose our goal is to make change using the smallest number of coins. The natural greedy algorithm iteratively chooses the largest denomination coin that is less than or equal to the amount of change that remains to be made. If we try this for 48 pence, we get three coins—30 + 12 + 6. However, the optimum answer would be two coins—24 + 24.

In its most general form, the coin changing problem is NP-hard on Page 37, but for some coinages, the greedy algorithm is optimum—e.g., if the denominations are of the form $\{1, r, r^2, r^3\}$. (An *ad hoc* argument can be applied to show that the greedy algorithm is also optimum for US coinage.) The general problem can be solved in pseudo-polynomial time using DP in a manner similar to Problem 17.6 on Page 285.

As another example of how greedy reasoning can fail, consider the following problem: Four travelers need to cross a river as quickly as possible in a small boat. Only two people can cross at one time. The time to cross the river is dictated by the slower person in the boat (if there is just one person, that is his time). The four travelers have times of 5, 10, 20, and 25 minutes. The greedy schedule would entail having the two fastest travelers cross initially (10), with the fastest returning (5), picking up the faster of the two remaining and crossing again (20), and with the fastest returning for the slowest traveler (5 + 25). The total time taken would be 10 + 5 + 20 + 5 + 25 = 65 minutes. However, a better approach would be for the fastest two to cross (10), with the faster traveler returning (5), and then having the two slowest travelers cross (25), with the second fastest returning (10) to pick up the fastest traveler (10). The total time for this schedule is 10 + 5 + 25 + 10 + 10 = 60 minutes.

Greedy algorithms boot camp

For US currency, wherein coins take values $1, 5, 10, 25, 50, 100$ cents, the greedy algorithm for making change results in the minimum number of coins. Here is an implementation of this algorithm. Note

that once it selects the number of coins of a particular value, it never changes that selection; this is the hallmark of a greedy algorithm.

```
int ChangeMaking(int cents) {
  const array<int, 6> kCoins = {100, 50, 25, 10, 5, 1};
  int num_coins = 0;
  for (int i = 0; i < kCoins.size(); i++) {
    num_coins += cents / kCoins[i];
    cents %= kCoins[i];
  }
  return num_coins;
}
```

We perform 6 iterations, and each iteration does a constant amount of computation, so the time complexity is $O(1)$.

A greedy algorithm is often the right choice for an **optimization problem** where there's a natural set of **choices to select from**. [Problem 18.1]

It's often easiest to conceptualize a greedy algorithm recursively, and then **implement** it using iteration for higher performance. [Problem 25.34]

Even if the greedy approach does not yield an optimum solution, it can give insights into the optimum algorithm, or serve as a heuristic.

Sometimes the right greedy choice is **not obvious**. [Problem 4.2]

18.1 Compute an optimum assignment of tasks

We consider the problem of assigning tasks to workers. Each worker must be assigned exactly two tasks. Each task takes a fixed amount of time. Tasks are independent, i.e., there are no constraints of the form "Task 4 cannot start before Task 3 is completed." Any task can be assigned to any worker.

We want to assign tasks to workers so as to minimize how long it takes before all tasks are completed. For example, if there are 6 tasks whose durations are $5, 2, 1, 6, 4, 4$ hours, then an optimum assignment is to give the first two tasks (i.e., the tasks with duration 5 and 2) to one worker, the next two (1 and 6) to another worker, and the last two tasks (4 and 4) to the last worker. For this assignment, all tasks will finish after $\max(5 + 2, 1 + 6, 4 + 4) = 8$ hours.

Design an algorithm that takes as input a set of tasks and returns an optimum assignment.

Hint: What additional task should be assigned to the worker who is assigned the longest task?

Solution: Simply enumerating all possible sets of pairs of tasks is not feasible—there are too many of them. (The precise number assignments is $\binom{n}{2}\binom{n-2}{2}\binom{n-4}{2}\ldots\binom{4}{2}\binom{2}{2} = n!/2^{n/2}$, where n is the number of tasks.)

Instead we should look more carefully at the structure of the problem. Extremal values are important—the task that takes the longest needs the most help. In particular, it makes sense to pair the task with longest duration with the task of shortest duration. This intuitive observation can be understood by looking at any assignment in which a longest task is not paired with a shortest task. By swapping the task that the longest task is currently paired with with the shortest task, we get an assignment which is at least as good.

Note that we are not claiming that the time taken for the optimum assignment is the sum of the maximum and minimum task durations. Indeed this may not even be the case, e.g., if the two

by installing cameras at the center of the castle that look out to the perimeter. Each camera can look along a ray. To save cost, you would like to minimize the number of cameras. See Figure 18.1 on the preceding page for an example.

Variant: There are a number of points in the plane that you want to observe. You are located at the point $(0, 0)$. You can rotate about this point, and your field-of-view is a fixed angle. Where direction should you face to maximize the number of visible points?

Invariants

An invariant is a condition that is true during execution of a program. Invariants can be used to design algorithms as well as reason about their correctness. For example, binary search, maintains the invariant that the space of candidate solutions contains all possible solutions as the algorithm executes.

Sorting algorithms nicely illustrate algorithm design using invariants. For example, intuitively, selection sort is based on finding the smallest element, the next smallest element, etc. and moving them to their right place. More precisely, we work with successively larger subarrays beginning at index 0, and preserve the invariant that these subarrays are sorted, their elements are less than or equal to the remaining elements, and the entire array remains a permutation of the original array.

As a more sophisticated example, consider Solution 15.7 on Page 239, specifically the $O(k)$ algorithm for generating the first k numbers of the form $a + b\sqrt{2}$. The key idea there is to process these numbers in sorted order. The queues in that code maintain multiple invariants: queues are sorted, duplicates are never present, and the separation between elements is bounded.

Invariants boot camp

Suppose you were asked to write a program that takes as input a sorted array and a given value and determines if there are two entries in the array that add up to that value. For example, if the array is $\langle -2, 1, 2, 4, 7, 11 \rangle$, then there are entries adding to 6 and to 10, but not to 0 and 13.

There are several ways to solve this problem: iterate over all pairs, or for each array entry search for the given value minus that entry. The most efficient approach uses invariants: maintain a subarray that is guaranteed to hold a solution, if it exists. This subarray is initialized to the entire array, and iteratively shrunk from one side or the other. The shrinking makes use of the sortedness of the array. Specifically, if the sum of the leftmost and the rightmost elements is less than the target, then the leftmost element can never be combined with some element to obtain the target. A similar observation holds for the rightmost element.

```
bool HasTwoSum(vector<int> A, int t) {
  int i = 0, j = A.size() - 1;
  while (i <= j) {
    if (A[i] + A[j] == t) {
      return true;
    } else if (A[i] + A[j] < t) {
      ++i;
    } else {  // A[i] + A[j] > t.
      --j;
    }
  }
  return false;
}
```

The time complexity is $O(n)$, where n is the length of the array. The space complexity is $O(1)$, since the subarray can be represented by two variables.

The key strategy to determine whether to use an invariant when designing an algorithm is to work on **small examples** to hypothesize the invariant. [Problems 18.4 and 18.6]

Often, the invariant is a subset of the set of input space, e.g,. a subarray. [Problems 18.4 and 18.7]

18.4 The 3-sum problem

Design an algorithm that takes as input an array and a number, and determines if there are three entries in the array (not necessarily distinct) which add up to the specified number. For example, if the array is $\langle 11, 2, 5, 7, 3\rangle$ then there are three entries in the array which add up to 21 ($3, 7, 11$ and $5, 5, 11$). (Note that we can use 5 twice, since the problem statement said we can use the same entry more than once.) However, no three entries add up to 22.

Hint: How would you check if a given array entry can be added to two more entries to get the specified number?

Solution: The brute-force algorithm is to consider all possible triples, e.g., by three nested for-loops iterating over all entries. The time complexity is $O(n^3)$, where n is the length of the array, and the space complexity is $O(1)$.

Let A be the input array and t the specified number. We can improve the time complexity to $O(n^2)$ by storing the array entries in a hash table first. Then we iterate over pairs of entries, and for each $A[i] + A[j]$ we look for $t - (A[i] + A[j])$ in the hash table. The space complexity now is $O(n)$.

We can avoid the additional space complexity by first sorting the input. Specifically, sort A and for each $A[i]$, search for indices j and k such that $A[j] + A[k] = t - A[i]$. We can do each such search in $O(n \log n)$ time by iterating over $A[j]$ values and doing binary search for $A[k]$.

We can improve the time complexity to $O(n)$ by starting with $A[0] + A[n-1]$. If this equals $t - A[i]$, we're done. Otherwise, if $A[0] + A[n-1] < t - A[i]$, we move to $A[1] + A[n-1]$—there is no chance of $A[0]$ pairing with any other entry to get $t - A[i]$ (since $A[n-1]$ is the largest value in A). Similarly, if $A[0] + A[n-1] > t - A[i]$, we move to $A[0] + A[n-2]$. This approach eliminates an entry in each iteration, and spends $O(1)$ time in each iteration, yielding an $O(n)$ time bound to find $A[j]$ and $A[k]$ such that $A[j] + A[k] = t - A[i]$, if such entries exist. The invariant is that if two elements which sum to the desired value exist, they must lie within the subarray currently under consideration.

For the given example, after sorting the array is $\langle 2, 3, 5, 7, 11\rangle$. For entry $A[0] = 2$, to see if there are $A[j]$ and $A[k]$ such that $A[0]+A[j]+A[k] = 21$, we search for two entries that add up to $21-2 = 19$.

The code for this approach is shown below.

```
bool HasThreeSum(vector<int> A, int t) {
  sort(A.begin(), A.end());

  for (int a : A) {
    // Finds if the sum of two numbers in A equals to t - a.
    if (HasTwoSum(A, t - a)) {
      return true;
    }
  }
  return false;
}
```

Chapter

19 Graphs

Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he would cross each bridge once and only once.

— *"The solution of a problem relating to the geometry of position,"* L. Euler, 1741

Informally, a graph is a set of vertices and connected by edges. Formally, a directed graph is a set V of *vertices* and a set $E \subset V \times V$ of edges. Given an edge $e = (u, v)$, the vertex u is its *source*, and v is its *sink*. Graphs are often decorated, e.g., by adding lengths to edges, weights to vertices, a start vertex, etc. A directed graph can be depicted pictorially as in Figure 19.1.

A *path* in a directed graph from u to vertex v is a sequence of vertices $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ where $v_0 = u$, $v_{n-1} = v$, and each (v_i, v_{i+1}) is an edge. The sequence may consist of a single vertex. The *length* of the path $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ is $n - 1$. Intuitively, the length of a path is the number of edges it traverses. If there exists a path from u to v, v is said to be *reachable* from u. For example, the sequence $\langle a, c, e, d, h \rangle$ is a path in the graph represented in Figure 19.1.

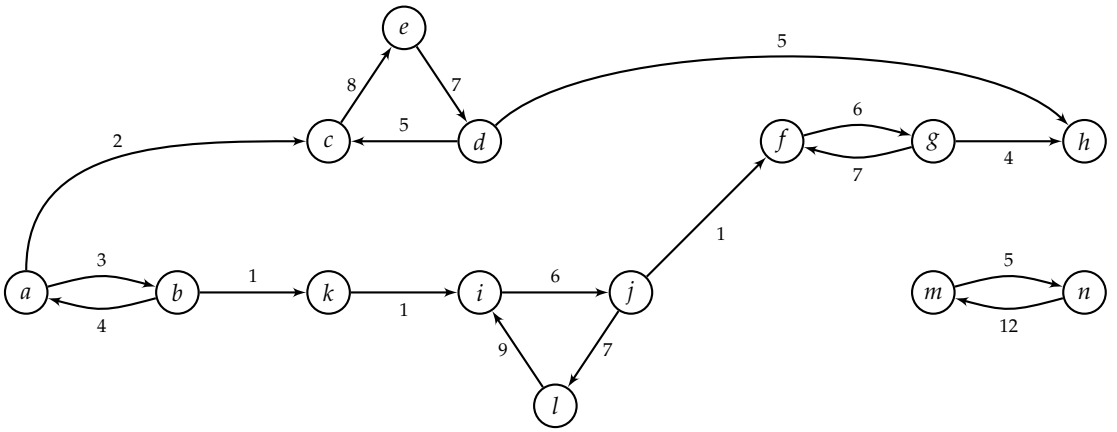


Figure 19.1: A directed graph with weights on edges.

A *directed acyclic graph* (*DAG*) is a directed graph in which there are no *cycles*, i.e., paths which contain one or more edges and which begin and end at the same vertex. See Figure 19.2 on the next page for an example of a directed acyclic graph. Vertices in a DAG which have no incoming edges are referred to as *sources*; vertices which have no outgoing edges are referred to as *sinks*. A *topological ordering* of the vertices in a DAG is an ordering of the vertices in which each edge is from a vertex earlier in the ordering to a vertex later in the ordering. Solution 19.8 on Page 330 uses the notion of topological ordering.

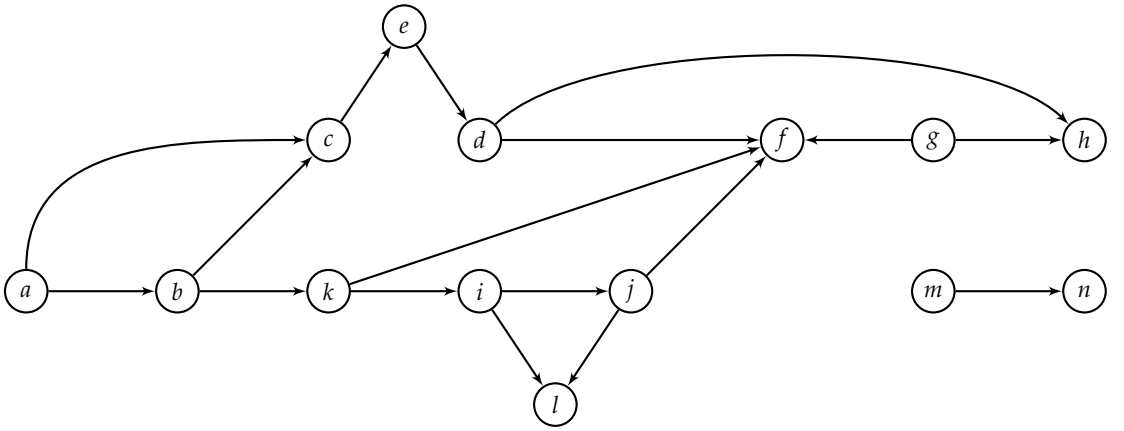


Figure 19.2: A directed acyclic graph. Vertices a, g, m are sources and vertices l, f, h, n are sinks. The ordering $\langle a, b, c, e, d, g, h, k, i, j, f, l, m, n \rangle$ is a topological ordering of the vertices.

An undirected graph is also a tuple (V, E); however, E is a set of unordered pairs of vertices. Graphically, this is captured by drawing arrowless connections between vertices, as in Figure 19.3.

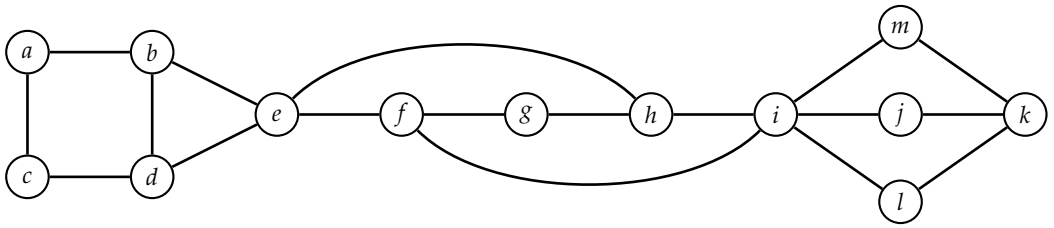


Figure 19.3: An undirected graph.

If G is an undirected graph, vertices u and v are said to be *connected* if G contains a path from u to v; otherwise, u and v are said to be *disconnected*. A graph is said to be connected if every pair of vertices in the graph is connected. A *connected component* is a maximal set of vertices C such that each pair of vertices in C is connected in G. Every vertex belongs to exactly one connected component.

For example, the graph in Figure 19.3 is connected, and it has a single connected component. If edge (h, i) is removed, it remains connected. If additionally (f, i) is removed, it becomes disconnected and there are two connected components.

A directed graph is called *weakly connected* if replacing all of its directed edges with undirected edges produces an undirected graph that is connected. It is *connected* if it contains a directed path from u to v or a directed path from v to u for every pair of vertices u and v. It is *strongly connected* if it contains a directed path from u to v and a directed path from v to u for every pair of vertices u and v.

Graphs naturally arise when modeling geometric problems, such as determining connected cities. However, they are more general, and can be used to model many kinds of relationships.

A graph can be implemented in two ways—using *adjacency lists* or an *adjacency matrix*. In the adjacency list representation, each vertex v, has a list of vertices to which it has an edge. The adjacency matrix representation uses a $|V| \times |V|$ Boolean-valued matrix indexed by vertices, with a 1 indicating the presence of an edge. The time and space complexities of a graph algorithm are

usually expressed as a function of the number of vertices and edges.

A *tree* (sometimes called a free tree) is a special sort of graph—it is an undirected graph that is connected but has no cycles. (Many equivalent definitions exist, e.g., a graph is a free tree if and only if there exists a unique path between every pair of vertices.) There are a number of variants on the basic idea of a tree. A rooted tree is one where a designated vertex is called the root, which leads to a parent-child relationship on the nodes. An ordered tree is a rooted tree in which each vertex has an ordering on its children. Binary trees, which are the subject of Chapter 10, differ from ordered trees since a node may have only one child in a binary tree, but that node may be a left or a right child, whereas in an ordered tree no analogous notion exists for a node with a single child. Specifically, in a binary tree, there is position as well as order associated with the children of nodes.

As an example, the graph in Figure 19.4 is a tree. Note that its edge set is a subset of the edge set of the undirected graph in Figure 19.3 on the preceding page. Given a graph $G = (V, E)$, if the graph $G' = (V, E')$ where $E' \subset E$, is a tree, then G' is referred to as a spanning tree of G.

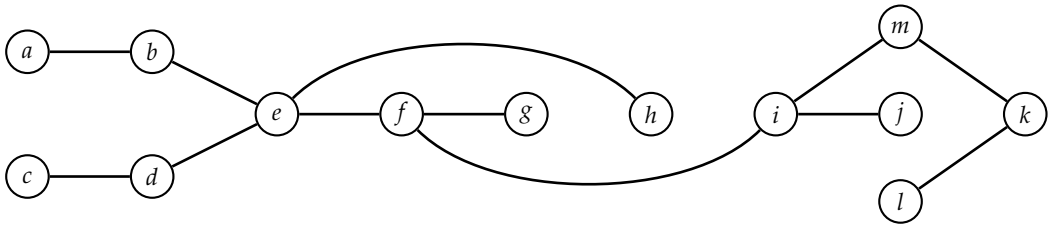


Figure 19.4: A tree.

Graphs boot camp

Graphs are ideal for modeling and analyzing relationships between pairs of objects. For example, suppose you were given a list of the outcomes of matches between pairs of teams, with each outcome being a win or loss. A natural question is as follows: given teams A and B, is there a sequence of teams starting with A and ending with B such that each team in the sequence has beaten the next team in the sequence?

A slick way of solving this problem is to model the problem using a graph. Teams are vertices, and an edge from one team to another indicates that the team corresponding to the source vertex has beaten the team corresponding to the destination vertex. Now we can apply graph reachability to perform the check. Both DFS and BFS are reasonable approaches—the program below uses DFS.

```
struct MatchResult {
  string winning_team, losing_team;
};

bool CanTeamABeatTeamB(const vector<MatchResult>& matches, const string& team_a,
                       const string& team_b) {
  return IsReachableDFS(BuildGraph(matches), team_a, team_b,
                        make_unique<unordered_set<string>>().get());
}

unordered_map<string, unordered_set<string>> BuildGraph(
    const vector<MatchResult>& matches) {
  unordered_map<string, unordered_set<string>> graph;
  for (const MatchResult& match : matches) {
    graph[match.winning_team].emplace(match.losing_team);
  }
```

```
  return graph;
}

bool IsReachableDFS(const unordered_map<string, unordered_set<string>>& graph,
                    const string& curr, const string& dest,
                    unordered_set<string>* visited_ptr) {
  unordered_set<string>& visited = *visited_ptr;
  if (curr == dest) {
    return true;
  } else if (visited.find(curr) != visited.end() ||
             graph.find(curr) == graph.end()) {
    return false;
  }
  visited.emplace(curr);
  for (const string& team : graph.at(curr)) {
    if (IsReachableDFS(graph, team, dest, visited_ptr)) {
      return true;
    }
  }
  return false;
}
```

The time complexity and space complexity are both $O(E)$, where E is the number of outcomes.

It's natural to use a graph when the problem involves **spatially connected** objects, e.g., road segments between cities. [Problems 19.1 and 19.2]

More generally, consider using a graph when you need to analyze **any binary relationship**, between objects, such as interlinked webpages, followers in a social graph, etc. [Problems 19.7 and 19.8]

Some graph problems entail **analyzing structure**, e.g., looking for cycles or connected components. **DFS** works particularly well for these applications. [Problem 19.4]

Some graph problems are related to **optimization**, e.g., find the shortest path from one vertex to another. **BFS, Dijkstra's shortest path algorithm, and minimum spanning tree** are examples of graph algorithms appropriate for optimization problems. [Problem 19.9]

Graph search

Computing vertices which are reachable from other vertices is a fundamental operation which can be performed in one of two idiomatic ways, namely depth-first search (DFS) and breadth-first search (BFS). Both have linear time complexity—$O(|V| + |E|)$ to be precise. In the worst-case there is a path from the initial vertex covering all vertices without any repeats, and the DFS edges selected correspond to this path, so the space complexity of DFS is $O(|V|)$ (this space is implicitly allocated on the function call stack). The space complexity of BFS is also $O(|V|)$, since in a worst-case there is an edge from the initial vertex to all remaining vertices, implying that they will all be in the BFS queue simultaneously at some point.

DFS and BFS differ from each other in terms of the additional information they provide, e.g., BFS can be used to compute distances from the start vertex and DFS can be used to check for the presence of cycles. Key notions in DFS include the concept of *discovery time* and *finishing time* for vertices.

Chapter

20 Parallel Computing

The activity of a computer must include the proper reacting to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in all information systems in which a number of computers are coupled to each other.

— "*Cooperating sequential processes*,"
E. W. Dijkstra, 1965

Parallel computation has become increasingly common. For example, laptops and desktops come with multiple processors which communicate through shared memory. High-end computation is often done using clusters consisting of individual computers communicating through a network.

Parallelism provides a number of benefits:

- High performance—more processors working on a task (usually) means it is completed faster.
- Better use of resources—a program can execute while another waits on the disk or network.
- Fairness—letting different users or programs share a machine rather than have one program run at a time to completion.
- Convenience—it is often conceptually more straightforward to do a task using a set of concurrent programs for the subtasks rather than have a single program manage all the subtasks.
- Fault tolerance—if a machine fails in a cluster that is serving web pages, the others can take over.

Concrete applications of parallel computing include graphical user interfaces (GUI) (a dedicated thread handles UI actions while other threads are, for example, busy doing network communication and passing results to the UI thread, resulting in increased responsiveness), Java virtual machines (a separate thread handles garbage collection which would otherwise lead to blocking, while another thread is busy running the user code), web servers (a single logical thread handles a single client request), scientific computing (a large matrix multiplication can be split across a cluster), and web search (multiple machines crawl, index, and retrieve web pages).

The two primary models for parallel computation are the shared memory model, in which each processor can access any location in memory, and the distributed memory model, in which a processor must explicitly send a message to another processor to access its memory. The former is more appropriate in the multicore setting and the latter is more accurate for a cluster. The questions in this chapter are focused on the shared memory model.

Writing correct parallel programs is challenging because of the subtle interactions between parallel components. One of the key challenges is races—two concurrent instruction sequences access the same address in memory and at least one of them writes to that address. Other challenges to correctness are

- starvation (a processor needs a resource but never gets it; e.g., Problem 20.6 on Page 342),
- deadlock (Thread A acquires Lock $L1$ and Thread B acquires Lock $L2$, following which A tries to acquire $L2$ and B tries to acquire $L1$), and
- livelock (a processor keeps retrying an operation that always fails).

Bugs caused by these issues are difficult to find using testing. Debugging them is also difficult because they may not be reproducible since they are usually load dependent. It is also often true that it is not possible to realize the performance implied by parallelism—sometimes a critical task cannot be parallelized, making it impossible to improve performance, regardless of the number of processors added. Similarly, the overhead of communicating intermediate results between processors can exceed the performance benefits.

The problems in this chapter focus on thread-level parallelism. Problems concerned with parallelism on distributed memory architectures, e.g., cluster computing, are usually not meant to be coded; they are popular design and architecture problems. Problems 21.9 on Page 355, 21.10 on Page 356, 21.11 on Page 356, and 21.17 on Page 361 cover some aspects of cluster-level parallelism.

Parallel computing boot camp

A semaphore is a very powerful synchronization construct. Conceptually, a semaphore maintains a set of permits. A thread calling *acquire()* on a semaphore waits, if necessary, until a permit is available, and then takes it. A thread calling *release()* on a semaphore adds a permit and notifies threads waiting on that semaphore, potentially releasing a blocking acquirer. The program below shows how to implement a semaphore in C++, using language primitives. (The C++ concurrency library provides a full-featured implementation of semaphores which should be used in practice.)

```
class Semaphore {
 public:
  Semaphore(int max_available) : max_available_(max_available), taken_(0) {}

  void Acquire() {
    unique_lock<mutex> lock(mx_);
    while (taken_ == max_available_) {
      cond_.wait(lock);
    }
    ++taken_;
  }

  void Release() {
    lock_guard<mutex> lock(mx_);
    --taken_;
    cond_.notify_all();
  }

 private:
  int max_available_;
  int taken_;
  mutex mx_;
  condition_variable cond_;
};
```

20.1 Implement caching for a multithreaded dictionary

The program below is part of an online spell correction service. Clients send as input a string, and the service returns an array of strings in its dictionary that are closest to the input string (this array could be computed, for example, using Solution 17.2 on Page 277). The service caches results to improve performance. Critique the implementation and provide a solution that overcomes its limitations.

```
class SpellCheckService {
```

Chapter

22 Language Questions

The limits of my language means the limits of my world.

— L. Wittgenstein

We now present some commonly asked problems on the C++ language. This set is not meant to be comprehensive—it's a test of preparedness. If you have a hard time with them, we recommend that you spend more time reviewing language.

22.1 References and pointers

What is a reference? How is it different from a pointer?
Solution: References and pointers are similar in that both refer to some other target object. However, there are key differences. Broadly, a pointer is a separate variable; a reference is an alias to its target.

- When a pointer's value is obtained, we receive the address of its target. To obtain the value of the target of `ptr`, we must dereference it (`*ptr`).
- When a reference's value is obtained, the dereferencing is performed automatically by code inserted by the compiler and we receive the target's value immediately. To obtain the address of `ref`, we must explicitly call the address-of operator (`&ref`).
- A pointer can be assigned an address of a different target and it can be assigned a `nullptr` value so that it doesn't point to any object. Since a pointer is a variable, it has an address of its own, which can be assigned to another variable, e.g., `int **intPtrPtr = &intPtr`.
- A reference can only be initialized, but not reassigned because assignment to a reference variable means assignment to the original object it refers to.

22.2 Pass-by-reference vs. pass-by-value

Why is it preferable to pass-by-reference rather than pass-by-value?
Solution: When an object is passed by value a new copy of it is created and passed to the function which may be an expensive operation, e.g., copying a large vector of objects. Furthermore, in some cases a class may preclude a copy being made of it, e.g., by making the copy constructor unavailable to clients by declaring it as private.

When passing by reference the object itself is passed; to be precise the function receives the object's address. In this case the function can modify the object, which may be the function's purpose. However, if it's undesirable to allow the function to modify the object, then the object must be passed as a const reference.

Another problem which might arise when passing by value is that an undesirable implicit type conversion may be performed, e.g., a derived class may be cast to base class which will prevent polymorphic behavior. For example, the following code will compile without any problems:

```
class Base {
 public:
  virtual string msg() { return "I am base"; }
};

class Child : public Base {
 public:
  virtual string msg() { return "I am child"; }
};

void cast(Base x) { cout << x.msg() << endl; }

int main() {
  Base f;
  Child b;
  cast(f);
  cast(b);
}
```

However, both calls to `cast()` will return "I am foo". If we change `cast()` to the following (leaving everything else the same)

```
void cast(Base& x) { cout << x.msg() << endl; }
```

Then the first call will return "I am bar" and the second call will return "I am foo".

22.3 Smart pointers

What is a smart pointer? What are the three commonly used smart pointer types? What is the appropriate use case for each one?

A smart pointer is a class which encapsulates an actual pointer to a dynamically allocated object, keeps track of its usage, and ensures that memory is deallocated when it is appropriate so that there is no memory/resources leak. The standard library includes these smart pointer classes:

- `unique_ptr`: destroys the object when the variable goes out of scope. May transfer ownership to another unique_ptr, e.g., when passing by value to a function. At all times there will be only one pointer referring to an object.
- `shared_ptr`: is freely copyable so that many variables can refer to the same object. Keeps track of the reference count and destroys the object when it is no longer in use.
- `weak_ptr`: refers to an object which is held by a `shared_ptr` but does not participate in reference counting so the object may be destroyed even if a `weak_ptr` refers to it. A `weak_ptr` is needed to solve the problem of reference cycles—if `shared_ptr` A points to B and `shared_ptr` B points to A, then neither will ever be destroyed.

Here is an example illustrating the difference between `shared_ptr` and `weak_ptr`.

```
struct Cycle1 {
  shared_ptr<Cycle2> next;
};

struct Cycle2 {
  shared_ptr<Cycle1> next;
};

auto head = make_shared<Cycle1>();  // head's reference count is now 1.
```

```
auto tail = make_shared<Cycle2>();  // tail's reference count is now 1.
head->next = tail;  // tail's reference count is now 2.
tail->next = head;  // head's reference count is now 2.
// On destruction of head and tail, the reference counts go to 1 and stay
// there.
```

When `head` and `tail` go out of scope the reference counts decrease by 1 so both ref counts will be 1. This is a leak, since `head` and `tail` can only be destroyed when their ref counts become 0. We can fix this leak by declaring `tail` to be of type `weak_ptr<Cycle2>`.

22.4 Iterators

In what ways are iterators similar to and different from pointers?
Solution: Both are similar in that you can derefence them to get a value. However, there are key differences.

- A pointer holds an address in memory. An iterator may hold a pointer, but it may be something much more complex. For example, an iterator can iterate over data that's on a file system, spread across many machines, or generated locally in a programmatic fashion. A good example is an iterator over a linked list—the iterator will move through elements that are at nodes in the list whose addresses in RAM may be scattered.
- You can perform simple arithmetic on pointers (increment, decrement, add a integer). Not all iterators allow these operations, e.g., you cannot decrement a forward-iterator, or add an integer to a nonrandom-access iterator.
- A pointer of type `T*` can point to any type `T` object. An iterator is more restricted, e.g., a `vector<double>::iterator` can only refer to doubles that are inside a `vector<double>` container.
- Since an iterator refers to objects in a container, unlike pointers, there's no concept of `delete` for an iterator. (The container is responsible for memory management.)

22.5 Constructors

What is a constructor? What is a default constructor? What is a copy constructor and how is it different from a move constructor?
Solution: In object-oriented programming, a constructor in a class is a special type of subroutine that is used to create an object. It prepares the new object for use—typically the constructor accepts arguments that it uses to set fields. The fields themselves may be objects, so a constructor may itself make calls to other constructors.

- The default constructor is a constructor which can be called with no arguments. It is called when an instance is created without initialization, e.g.,

  ```
  Foo x;  // x is initialized by the default constructor
  ```

  If no user-defined constructor exists for the class, the compiler will create a default constructor. The synthesized contstructor may be trivial, i.e., do nothing at all. Trivial constructors are generated when the class' fields are primitive types or classes that also have trivial constructors, and when the class does not subclass a base class or it subclasses classes whose constructors are trivial; in addition, it cannot have virtual functions or default initializers for members. Otherwise, if the class has fields that are themselves objects with nontrivial

constructors, the default constructor for these objects is called. If the class subclasses a base class with a nontrivial default constructor, the base class' default constructor is also called.

- The copy constructor is a constructor which can be called with a reference to a class instance as an argument, typically `ClassName(const ClassName&)`. It is called when a new copy of an instance needs to be initialized either explicitly as in `ClassName new_instance(existing_instance)` or implicitly by compiler, e.g., when an instance is passed by value to a function or is returned by value.
- The move constructor is a constructor which can be called with an rvalue reference to a class instance as an argument, typically `ClassName(ClassName&&)`. It is called when a new instance is initialized from a temporary object that typically is destroyed after initialization, e.g., when returning by value from a function or an explicit call as in `ClassName new_instance(std::move(existing_instance))`.

22.6 Default methods

If you write a class that has no methods on it, and the class does not inherit from another class, the compiler will add four methods to it automatically. What are these methods, and why should you be aware of them?

Solution: The methods which can be automatically generated by the compiler are:

- Default constructor: equivalent to an empty default constructor (Solution 22.5 on the preceding page).
- Destructor: equivalent to an empty destructor, calls the superclass destructor and the destructor for member fields that are not of primitive type.
- Copy constructor: equivalent to a copy constructor that initializes every instance member with a corresponding member of the constructor's argument. Note that this may lead to calls to the member field's own copy constructors.
- Copy assignment operator: equivalent to an assignment operator that assigns every member of its argument to a corresponding member of this instance. Note that this may lead to calls to the member field's own copy assignment operators.

It's important to be aware of these functions. The problem with them is not in their existence but that you are not forced to write your own and may forget to do so in cases where trivial copy/initialization will not work. For example, the following program prints 1 then −2. This is because the default copy assignment operator copies the raw pointer to `b2`.

```
class Buffer {
 public:
  Buffer(int size, int* buffer) : size(size), buffer(buffer) {}

  int size;
  int* buffer;
};

const int kBufSize = 2;
int* buffer = new int[kBufSize]{1, 2};

Buffer b1 = Buffer(kBufSize, buffer);
cout << b1.buffer[0] << endl;

Buffer b2 = b1;
```

```
b2.buffer[0] = -2;
cout << b1.buffer[0] << endl;
```

Similar problems abound when using the default constructor and destructor—fields are not initialized, memory is not reclaimed, etc. In short, you should be aware that copying and initializing objects is your responsibility.

C++11 introduces move semantics, which results in every class having two additional methods, a move copy constructor and a move copy assignment, that are synthesized by the compiler. (The actual situation is slightly more complex, e.g., if you write a copy assignment operator, the compiler will not synthesize the move copy assignment for you.)

22.7 malloc(), free(), new, delete

How are `malloc()/free()` similar to and different from `new/delete`?
Solution: Both are similar in that they allocate space on the heap, e.g., for objects and for arrays. However, there are major differences in the level of abstraction that they provide.

- `malloc()` is type-agnostic, it allocates an uninitialized memory block of required size in bytes and returns a void pointer.
- `new` allocates a memory block required to hold an object of specific type, initializes the object using appropriate the constructor and returns a type-safe pointer.
- A single call to `new` may lead to multiple additional calls to new depending on the nature of the fields of the object/array being allocated.
- `new` may throw an exception when no memory is available.
- `delete` besides freeing the memory ensures that object's destructor is called.
- It is common, though not essential, for `new` to use `malloc()` to allocate memory and for `delete` to use `free()` to deallocate it.

22.8 Strings

What are the differences between C-style strings and C++ strings?
Solution: Both are similar in that they are used to manipulate character sequences.

A C string is not a type of its own but an array of chars terminated by convention with the character `'\0'`. A C++ string is an object from the `string` class.

There are a number of library methods for C strings, which use the fact that the string is null terminated, e.g., `strlen(s)`, `strcat(s,t)`, `strrchr(str,'s')`. None of these have any concept of safety—if the argument is not null terminated, they may crash with out-of-bound access errors, or silently corrupt memory. With C strings, the caller is responsible for the size of the destination string.

C++ `string` methods have a much richer set of operators.

- Range-based looping can be performed over the characters in a string.
- Strings can be initialized in a variety of ways, e.g., `string t(u, 2, 4)`.
- Strings can be updated in many ways, e.g., `s.insert(s.size(), 3, '!')`, and `s.erase(11, 5)`.
- Strings can be searched in rich ways, e.g., `name.find_first_of("0123")`, and `river.rfind("ssi")`.
- The + operators can be applied to strings, e.g., `s = "Carl" + g`.

- The comparison operators $<, \leq, ==, \geq, >$ can be applied to strings, with == testing logical equality, rather than pointer equality (as is the case for C strings, which must be tested for logical equality using `strcmp(s,t) == 0`).

C++ `string` methods also have some support for range checking, and thus are safer.

22.9 `push_back()` and `emplace_back()`

What is the difference between `vs.push_back("xyzzy")` and `vs.emplace_back("xyzzy")` Which is preferable?

Solution: Operationally, both are similar in that they add entries to the back of a vector. They differ significantly in terms of how copying/moving is actually implemented, which has implications to performance.

- `vs.push_back(s)` copies a string into a vector. First, a new string object will be implicitly created initialized with provided `char*`. Then `push_back` will be called which will copy this string into the vector using the move constructor because the original string is a temporary object. Then the temporary object will be destroyed.
- `vs.emplace_back(s)` constructs a string in-place, so no temporary string will be created but rather `emplace_back` will be called directly with `char*` argument. It will then create a string to be stored in the vector initialized with this `char*`. So in this case we avoid constructing and destroying an unnecessary temporary string object

22.10 Updating a map

Criticize the following two attempts to update a key that belongs to a map and propose a solution that works.

```
// Attempt 1
unordered_map<Point, string, HashPoint> table;
Point p{1, 2};
table[p] = "Minkowski";
p.x = 3;
```

```
// Attempt 2
unordered_map<Point, string, HashPoint> table;
Point p{1, 2};
table[p] = "Minkowski";
auto iter = table.find(p);
iter->first.x = 3;
```

Solution: The code in Attempt 1 doesn't change the key's value: the p object is copied into table, changing it will not change the key as it's a separate instance. The code in Attempt 2 will not even compile, since `iter->first` is const.

A correct way to perform the update is as follows.

```
Point p(1, 2);
unordered_map<Point, string, HashPoint> table;
table[p] = "Minkowski";
auto val = table[p];
table.erase(p);
p.x = 4;
table[p] = val;
```

However, this approach involves a wasteful copy to `val`. A much more efficient approach when dealing with keys that are large objects is as follows:

```
Point p(1, 2);
unordered_map<Point, string, HashPoint> table;
table[p] = "Minkowski";
auto it = table.find(p);
p.x = 4;
std::swap(table[p], it->second);
table.erase(it);
```

An equivalent approach is as follows:

```
Point p(1, 2);
unordered_map<Point, string, HashPoint> table;
table[p] = "Minkowski";
auto it = table.find(p);
p.x = 4;
table[p] = std::move(it->second);
table.erase(it);
```

22.11 Fast function calls

A developer wrote the following macro to test if a character is a vowel. Criticize his approach, and suggest an improvement.

```
// Avoid overhead of a function call.
#define isvowel(c) (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
```

Solution: The developer's goal is to get more performance by avoiding the cost of a function call which entails setting up the call stack, passing arguments, and setting return address.

Macros are a poor choice for this (and just about everything else). Compared to a function call, there's limited/no typechecking; they can lead to bloat in the object code; and debugging them can be hard.

A bigger issue is that they can lead to very nasty bugs: for example if we were to write `isvowel(aChar++)`, depending on what `aChar` is, it could get incremented from one to five times, since the macro expands to `(aChar++ == 'a' || aChar++ == 'e' || aChar++ == 'i' || aChar++ == 'o' || aChar++ == 'u')`.

Writing `isvowel()` as a function overcomes these limitations. Furthermore, we can achieve the same performance by telling the compiler to inline the function:

```
inline bool isvowel(char c) {
  return (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
}
```

22.12 Template functions

Write a template function that takes two arguments and swaps them if the first is smaller than the second. Optimize your function for the case where the arguments may be very large objects.

Solution: Here is a straightforward use of templates. It will work with any type that implements a `bool operator>(T y)` method. (If used with a type that does not implement such a function, the compiler will issue an error to this effect.)

```
template <typename T>
void minmax(T& x, T& y) {
  if (x > y) {
    T tmp = x;
    x = y;
    y = tmp;
  }
}
```

Its primary shortcoming is that it uses three copies to achieve the transfer. We can do better using a move constructor.

```
template <typename T>
void minmax(T& x, T& y) {
  if (x > y) {
    T tmp = std::move(x);
    x = std::move(y);
    y = std::move(tmp);
    // Alternative: call std::swap(x, y)
  }
}
```

The first function takes 6000 ms when run 10^6 times on vector of length 10^4; the second one takes 5 ms.

22.13 Run-time type identification

What is the following code doing? Can you suggest a way to improve it?

```
class A {
 public:
  virtual void foo() { cout << "A's foo" << endl; }
};

class B : public A {
 public:
  void foo() override { cout << "B's foo" << endl; }
  virtual void bar() { cout << "B's bar" << endl; }
};

class C : public B {
 public:
  void bar() override { cout << "C's bar" << endl; }
  void widget() { cout << "C's widget" << endl; }
};

void bad(A* x) {
  if (typeid(*x) == typeid(A)) {
    x->foo();
  } else if (typeid(*x) == typeid(B)) {
    ((B*)x)->foo();
    ((B*)x)->bar();
  } else if (typeid(*x) == typeid(C)) {
    ((C*)x)->foo();
    ((C*)x)->bar();
    ((C*)x)->widget();
```

```
  }
}

int main() {
  // Randomly returns a pointer to  an A, B, or C type object.
  A* x = randomBuilder();
  bad(x);
}
```

Solution: This program is checking the runtime type of the argument to determine what methods it supports. For example, if the runtime type of x is *C*, it will print

```
B's foo
C's bar
C's widget
```

Its shortcoming is that it is not flexible. For example, if we were to add a new derived type, e.g., *D*, which subclasses *B*, but does not add additional methods, we'd still have to change the code to the following:

```
class D : public B {
 public:
  void bar() override { cout << "D's bar" << endl; }
};

void bad(A* x) {
  if (typeid(*x) == typeid(A)) {
    x->foo();
  } else if (typeid(*x) == typeid(B) || typeid(*x) == typeid(D)) {
    ((B*)x)->foo();
    ((B*)x)->bar();
  } else if (typeid(*x) == typeid(C)) {
    ((C*)x)->foo();
    ((C*)x)->bar();
    ((C*)x)->widget();
  }
}
```

A more maintainable solution is to dynamic casting, which allows us to inspect the runtime type of the object. Note that the function below works equally well when we add *D* as a subclass of *B*.

```
void good(A* x) {
  x->foo();
  // Objects of type D can be cast to a B, so the call below is to D's bar().
  B* p = dynamic_cast<B*>(x);
  if (p) {
    p->bar();
  }
  C* q = dynamic_cast<C*>(x);
  if (q) {
    q->widget();
  }
}
```

Explain what dynamic linkage is in the context of a C++ program, including its advantages and disadvantages.

Solution: Briefly, the C++ compiler takes C++ source files and creates object files from them. Object files are not executable because addresses of external functions and global variables remain to be filled in.

Conceptually, the linker takes object code and libraries (which are archives of object files) and assembles them into a single executable. In particular, it resolves all addresses to function calls and global variables.

Linking can be static, i.e., the output is a single program, conceptually a sequence of machine instructions. However, it is far more common to use dynamic linkage, wherein library code, such as sorting routines, typed input-output, etc. is added to the program when the process starts. (It's also possible, though uncommon, to load the library after the program has started executing; this is referred to as dynamic loading.)

- The benefits of dynamic linkage are smaller executables and not having to update the binary every time the library is updated.
- The key disadvantage of dynamic linkage is incompatibility when a library is updated independently and there's nobody to check and enforce correct interaction. In particular, this means a program that worked today may be unchanged tomorrow, but stop working. Other minor disadvantages are potentially slightly more runtime overhead when the library is loaded.

Chapter

23 Object-Oriented Design

One thing expert designers know not to do is solve every problem from first principles.

— "*Design Patterns: Elements of Reusable Object-Oriented Software*," E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, 1994

A class is an encapsulation of data and methods that operate on that data. Classes match the way we think about computation. They provide encapsulation, which reduces the conceptual burden of writing code, and enable code reuse, through the use of inheritance and polymorphism. However, naive use of object-oriented constructs can result in code that is hard to maintain.

A design pattern is a general repeatable solution to a commonly occurring problem. It is not a complete design that can be coded up directly—rather, it is a description of how to solve a problem that arises in many different situations. In the context of object-oriented programming, design patterns address both reuse and maintainability. In essence, design patterns make some parts of a system vary independently from the other parts.

Adnan's Design Pattern course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

23.1 Template Method vs. Strategy

Explain the difference between the template method pattern and the strategy pattern with a concrete example.

Solution: Both the template method and strategy patterns are similar in that both are behavioral patterns, both are used to make algorithms reusable, and both are general and very widely used. However, they differ in the following key way:

- In the template method, a skeleton algorithm is provided in a superclass. Subclasses can override methods to specialize the algorithm.
- The strategy pattern is typically applied when a family of algorithms implement a common interface. These algorithms can then be selected by clients.

As a concrete example, consider a sorting algorithm like quicksort. Two of the key steps in quicksort are pivot selection and partitioning. Quicksort is a good example of a template method—subclasses can implement their own pivot selection algorithm, e.g., using randomized median finding or selecting an element at random, and their own partitioning method, e.g., using the DNF partitioning algorithms in Solution 6.1 on Page 56.

Since there may be multiple ways in which to sort elements, e.g., student objects may be compared by GPA, major, name, and combinations thereof, it's natural to make the comparision operation used by the sorting algorithm an argument to quicksort. One way to do this is to pass quicksort an object that implements a compare method. These objects constitute an example of the strategy pattern, as do the objects implementing pivot selection and partitioning.

There are some other smaller differences between the two patterns. For example, in the template method pattern, the superclass algorithm may have "hooks"—calls to placeholder methods that can be overridden by subclasses to provide additional functionality. Sometimes a hook is not implemented, thereby forcing the subclasses to implement that functionality; some times it offers a "no-operation" or some baseline functionality. There is no analog to a hook in a strategy pattern.

Note that there's no relationship between the template method pattern and template meta-programming (a form of generic programming favored in C++).

23.2 Observer pattern

Explain the observer pattern with an example.

Solution: The observer pattern defines a one-to-many dependency between objects so that when one object changes state all its dependents are notified and updated automatically.

The observed object must implement the following methods.

- Register an observer.
- Remove an observer.
- Notify all currently registered observers.

The observer object must implement the following method.

- Update the observer. (Update is sometimes referred to as notify.)

As a concrete example, consider a service that logs user requests, and keeps track of the 10 most visited pages. There may be multiple client applications that use this information, e.g., a leaderboard display, ad placement algorithms, recommendation system, etc. Instead of having the clients poll the service, the service, which is the observed object, provides clients with register and remove capabilities. As soon as its state changes, the service enumerates through registered observers, calling each observer's update method.

Though most readily understood in the context of a single program, where the observed and observer are objects, the observer pattern is also applicable to distributed computing.

23.3 Push vs. pull observer pattern

In the observer pattern, subjects push information to their observers. There is another way to update data—the observers may "pull" the information they need from the subject. Compare and contrast these two approaches.

Solution: Both push and pull observer designs are valid and have tradeoffs depending on the needs of the project. With the push design, the subject notifies the observer that their data is ready and includes the relevant information that the observer is subscribing to, whereas with the pull design, it is the observer's job to retrieve that information from the subject.

The pull design places a heavier load on the observers, but it also allows the observer to query the subject only as often as is needed. One important consideration is that by the time the observer retrieves the information from the subject, the data could have changed. This could be a positive or negative result depending on the application. The pull design places less responsibility on the subject for tracking exactly which information the observer needs, as long as the subject knows when to notify the observer. This design also requires that the subject make its data publicly accessible by the observers. This design would likely work better when the observers are running with varied frequency and it suits them best to get the data they need on demand.

The push design leaves all of the information transfer in the subject's control. The subject calls update for each observer and passes the relevant information along with this call. This design seems more object-oriented, because the subject is pushing its own data out, rather than making its data accessible for the observers to pull. It is also somewhat simpler and safer in that the subject always knows when the data is being pushed out to observers, so you don't have to worry about an observer pulling data in the middle of an update to the data, which would require synchronization.

23.4 Singletons and Flyweights

Explain the differences between the singleton pattern and the flyweight pattern. Use concrete examples.

Solution: The singleton pattern ensures a class has only one instance, and provides a global point of access to it. The flyweight pattern minimizes memory use by sharing as much data as possible with other similar objects. It is a way to use objects in large numbers when a simple repeated representation would use an unacceptable amount of memory.

A common example of a singleton is a logger. There may be many clients who want to listen to the logged data (console, file, messaging service, etc.), so all code should log to a single place.

A common example of a flyweight is string interning—a method of storing only one copy of each distinct string value. Interning strings makes some string processing tasks more time- or space-efficient at the cost of requiring more time when the string is created or interned. The distinct values are usually stored in a hash table. Since multiple clients may refer to the same flyweight object, for safety flyweights should be immutable.

There is a superficial similarity between singleton and flyweight: both keep a single copy of an object. There are several key differences between the two:

- Flyweights are used to save memory. Singletons are used to ensure all clients see the same object.
- A singleton is used where there is a single shared object, e.g., a database connection, server configurations, a logger, etc. A flyweight is used where there is a family of shared objects, e.g., objects describing character fonts, or nodes shared across multiple binary search trees.
- Flyweight objects are invariable immutable. Singleton objects are usually not immutable, e.g., requests can be added to the database connection object.
- The singleton pattern is a creational pattern, whereas the flyweight is a structural pattern.

In summary, a singleton is like a global variable, whereas a flyweight is like a pointer to a canonical representation.

Sometimes, but not always, a singleton object is used to create flyweights—clients ask the singleton for an object with specified fields, and the singleton checks its internal pool of flyweights to see if one exists. If such an object already exists, it returns that, otherwise it creates a new flyweight, add it to its pool, and then returns it. (In essence the singleton serves as a gateway to a static factory.)

23.5 Adapters

What is the difference between a class adapter and an object adapter?

Solution: The adapter pattern allows the interface of an existing class to be used from another interface. It is often used to make existing classes work with others without modifying their source code.

There are two ways to build an adapter: via subclassing (the class adapter pattern) and composition (the object adapter pattern). In the class adapter pattern, the adapter inherits both the interface that is expected and the interface that is pre-existing. In the object adapter pattern, the adapter contains an instance of the class it wraps and the adapter makes calls to the instance of the wrapped object.

Here are some remarks on the class adapter pattern.

- The class adapter pattern allows re-use of implementation code in both the target and adaptee. This is an advantage in that the adapter doesn't have to contain boilerplate pass-throughs or cut-and-paste reimplementation of code in either the target or the adaptee.
- The class adapter pattern has the disadvantages of inheritance (changes in base class may cause unforeseen misbehaviors in derived classes, etc.). The disadvantages of inheritance are made worse by the use of two base classes, which also precludes its use in languages like Java (pre 1.8) that do not support multiple inheritance.
- The class adapter can be used in place of either the target or the adaptee. This can be advantage if there is a need for a two-way adapter. The ability to substitute the adapter for adaptee can be a disadvantage otherwise as it dilutes the purpose of the adapter and may lead to incorrect behavior if the adapter is used in an unexpected manner.
- The class adapter allows details of the behavior of the adaptee to be changed by overriding the adaptee's methods. Class adapters, as members of the class hierarchy, are tied to specific adaptee and target concrete classes.

As a concrete example of an object adapter, suppose we have legacy code that returns objects of type stack. Newer code expects inputs of type deque, which is more general than stack (but does not subclass stack). We could create a new type, stack-adapter, which implements the deque methods, and can be used anywhere deque is required. The stack-adapter class has a field of type stack—this is referred to as object composition. It implements the deque methods with code that uses methods on the composed stack object. Deque methods that are not supported by the underlying stack throw unsupported operation exceptions. In this scenario, the stack-adapter is an example of an object adapter.

Here are some comments on the object adapter pattern.

- The object adapter pattern is "purer" in its approach to the purpose of making the adaptee behave like the target. By implementing the interface of the target only, the object adapter is only useful as a target.
- Use of an interface for the target allows the adaptee to be used in place of any prospective target that is referenced by clients using that interface.
- Use of composition for the adaptee similarly allows flexibility in the choice of the concrete classes. If adaptee is a concrete class, any subclass of adaptee will work equally well within the object adapter pattern. If adaptee is an interface, any concrete class implementing that interface will work.
- A disadvantage is that if target is not based on an interface, target and all its clients may need to change to allow the object adapter to be substituted.

Variant: The UML diagrams for decorator, adapter, and proxy look identical, so why is each considered a separate pattern?

23.6 Creational Patterns

Explain what each of these creational patterns is: builder, static factory, factory method, and abstract factory.

Solution: The idea behind the builder pattern is to build a complex object in phases. It avoids mutability and inconsistent state by using an mutable inner class that has a build method that returns the desired object. Its key benefits are that it breaks down the construction process, and can give names to steps. Compared to a constructor, it deals far better with optional parameters and when the parameter list is very long.

A static factory is a function for construction of objects. Its key benefits are as follow: the function's name can make what it's doing much clearer compared to a call to a constructor. The function is not obliged to create a new object—in particular, it can return a flyweight. It can also return a subtype that's more optimized, e.g., it can choose to construct an object that uses an integer in place of a Boolean array if the array size is not more than the integer word size.

A factory method defines interface for creating an object, but lets subclasses decided which class to instantiate. The classic example is a maze game with two modes—one with regular rooms, and one with magic rooms. The program below uses a template method, as described in Problem 23.1 on Page 374, to combine the logic common to the two versions of the game.

```
class MazeGameCreator {
 public:
  virtual Room* MakeRoom() = 0;
  // This factory method is a template method for creating MazeGame objects.
  // MazeGameCreator's subclasses implement MakeRoom() as appropriate for
  // the type of room being created.
  MazeGame* FactoryMethod() {
    MazeGame* mazeGame = new MazeGame();
    Room* room1 = MakeRoom();
    Room* room2 = MakeRoom();
    room1->Connect(room2);
    mazeGame->AddRoom(room1);
    mazeGame->AddRoom(room2);
    return mazeGame;
  }
};
```

This snippet implements the regular rooms.

```
class OrdinaryMazeGameCreator : public MazeGameCreator {
  Room* MakeRoom() override { return new OrdinaryRoom(); }
};
```

This snippet implements the magic rooms.

```
class MagicMazeGameCreator : public MazeGameCreator {
  Room* MakeRoom() override { return new MagicRoom(); }
};
```

Here's how you use the factory to create regular and magic games.

```
MazeGame* ordinaryMazeGame = (new OrdinaryMazeGameCreator())->FactoryMethod();
MazeGame* magicMazeGame = (new MagicMazeGameCreator())->FactoryMethod();
```

A drawback of the factory method pattern is that it makes subclassing challenging.

An abstract factory provides an interface for creating families of related objects without specifying their concrete classes. For example, a class `DocumentCreator` could provide interfaces to create a number of products, such as `createLetter()` and `createResume()`. Concrete implementations of this class could choose to implement these products in different ways, e.g., with modern or classic

fonts, right-flush or right-ragged layout, etc. Client code gets a `DocumentCreator` object and calls its factory methods. Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. The price for this flexibility is more planning and upfront coding, as well as and code that may be harder to understand, because of the added indirections.

23.7 Libraries and design patterns

Why is there no library of design patterns so a developers do not have to write code every time they want to use them?

Solution: There are several reasons that design patterns cannot be delivered as a set of libraries. The key idea is that patterns cannot be cleanly abstracted from the objects and the processes they are applicable to. More specifically, libraries provide the implementations of algorithms. In contrast, design patterns provide a higher level understanding of how to structure classes and objects to solve specific types of problems. Another difference is that it's often necessary to use combinations of different patterns to solve a problem, e.g., Model-View-Controller (MVC), which is commonly used in UI design, incorporates the Observer, Strategy, and Composite patterns. It's not reasonable to come up with libraries for every possible case.

Of course, many libraries take advantage of design patterns in their implementations: sorting and searching algorithms use the template method pattern, custom comparision functions illustrate the strategy pattern, string interning is an example of the flyweight pattern, typed-I/O shows off the decorator pattern, etc.

Variant: Give examples of commonly used library code that use the following patterns: template, strategy, observer, singleton, flyweight, static factory, decorator, abstract factory.

Variant: Compare and contrast the iterator and composite patterns.

Chapter

24 Common Tools

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages.

— "*The UNIX TimeSharing System*,"
D. Ritchie and K. Thompson, 1974

The problems in this chapter are concerned with tools: version control systems, scripting languages, build systems, databases, and the networking stack. Such problems are not commonly asked—expect them only if you are interviewing for a specialized role, or if you claim specialized knowledge, e.g., network security or databases. We emphasize these are vast subjects, e.g., networking is taught as a sequence of courses in a university curriculum. Our goal here is to give you a flavor of what you might encounter. Adnan's Advanced Programming Tools course material, available freely online, contains lecture notes, homeworks, and labs that may serve as a good resource on the material in this chapter.

Version control

Version control systems are a cornerstone of software development—every developer should understand how to use them in an optimum way.

24.1 Merging in a version control system

What is merging in a version control system? Specifically, describe the limitations of line-based merging and ways in which they can be overcome.

Solution:

Modern version control systems allow developers to work concurrently on a personal copy of the entire codebase. This allows software developers to work independently. The price for this merging: periodically, the personal copies need to be integrated to create a new shared version. There is the possibility that parallel changes are conflicting, and these must be resolved during the merge.

By far, the most common approach to merging is text-based. Text-based merge tools view software as text. Line-based merging is a kind of text-based merging, specifically one in which each line is treated as an indivisible unit. The merging is "three-way": the lowest common ancestor in the revision history is used in conjunction with the two versions. With line-based merging of text files, common text lines can be detected in parallel modifications, as well as text lines that have

been inserted, deleted, modified, or moved. Figure 24.1 on the following page shows an example of such a line-based merge.

One issue with line-based merging is that it cannot handle two parallel modifications to the same line: the two modifications cannot be combined, only one of the two modifications must be selected. A bigger issue is that a line-based merge may succeed, but the resulting program is broken because of syntactic or semantic conflicts. In the scenario in Figure 24.1 on the next page the changes made by the two developers are made to independent lines, so the line-based merge shows no conflicts. However, the program will not compile because a function is called incorrectly.

In spite of this, line-based merging is widely used because of its efficiency, scalability,and accuracy. A three-way, line-based merge tool in practice will merge 90% of the changed files without any issues. The challenge is to automate the remaining 10% of the situations that cannot be merged automatically.

Text-based merging fails because it does not consider any syntactic or semantic information.

Syntactic merging takes the syntax of the programming language into account. Text-based merge often yields unimportant conflicts such as a code comment that has been changed by different developers or conflicts caused by code reformatting. A syntactic merge can ignore all these: it displays a merge conflict when the merged result is not syntactically correct.

We illustrate syntactic merging with a small example:

```
if (n%2 == 0) then m = n/2;
```

Two developers change this code in a different ways, but with the same overall effect. The first updates it to

```
if (n%2 == 0) then m = n/2 else m = (n-1)/2;
```

and the second developer's update is

```
m = n/2;
```

Both changes to the same thing. However, a textual-merge will likely result in

```
m := n div 2 else m := (n-1)/2
```

which is syntactically incorrect. Syntactic merge identifies the conflict; it is the integrator's responsibility to manually resolve it, e.g., by removing the `else` portion.

Syntactic merge tools are unable to detect some frequently occurring conflicts. For example, in the merge of Versions *1a* and *1b* in Figure 24.1 on the following page will not compile, since the call to `sum(10)` has the wrong signature. Syntactic merge will not detect this conflict since the program is still syntactically correct. The conflict is a semantic conflict, specifically, a static semantic conflict, since it is detectable at compile-time. (The compile will return something like "function argument mismatch error".)

Technically, syntactic merge algorithms operate on the parse trees of the programs to be merged. More powerful static semantic merge algorithms have been proposed that operate on a graph representation of the programs, wherein definitions and usage are linked, making it easier to identify mismatched usage.

Static semantic merging also has shortcomings. For example, suppose `Point` is a class representing $2D$-points using Cartesian coordinates, and that this class has a distance function that return $\sqrt{x^2 + y^2}$. Suppose Alice checks out out the project, and subclasses `Point` to create a class that supports polar coordinates, and uses `Points`'s distance function to return the radius. Concurrently, Bob checks out the project and changes `Point`'s distance function to return $|x| + |y|$. Static semantic

merging reports no conflicts: the merged program compiles without any trouble. However the behavior is not what was expected.

Both syntactic merging and semantic merging greatly increase runtimes, and are very limiting since they are tightly coupled to specific programming languages. In practice, line-based merging is used in conjunction with a small set of unit tests (a "smoke suite") invoked with a pre-commit hook script. Compilation finds syntax and static semantics errors, and the tests (hopefully) identify deeper semantic issues.

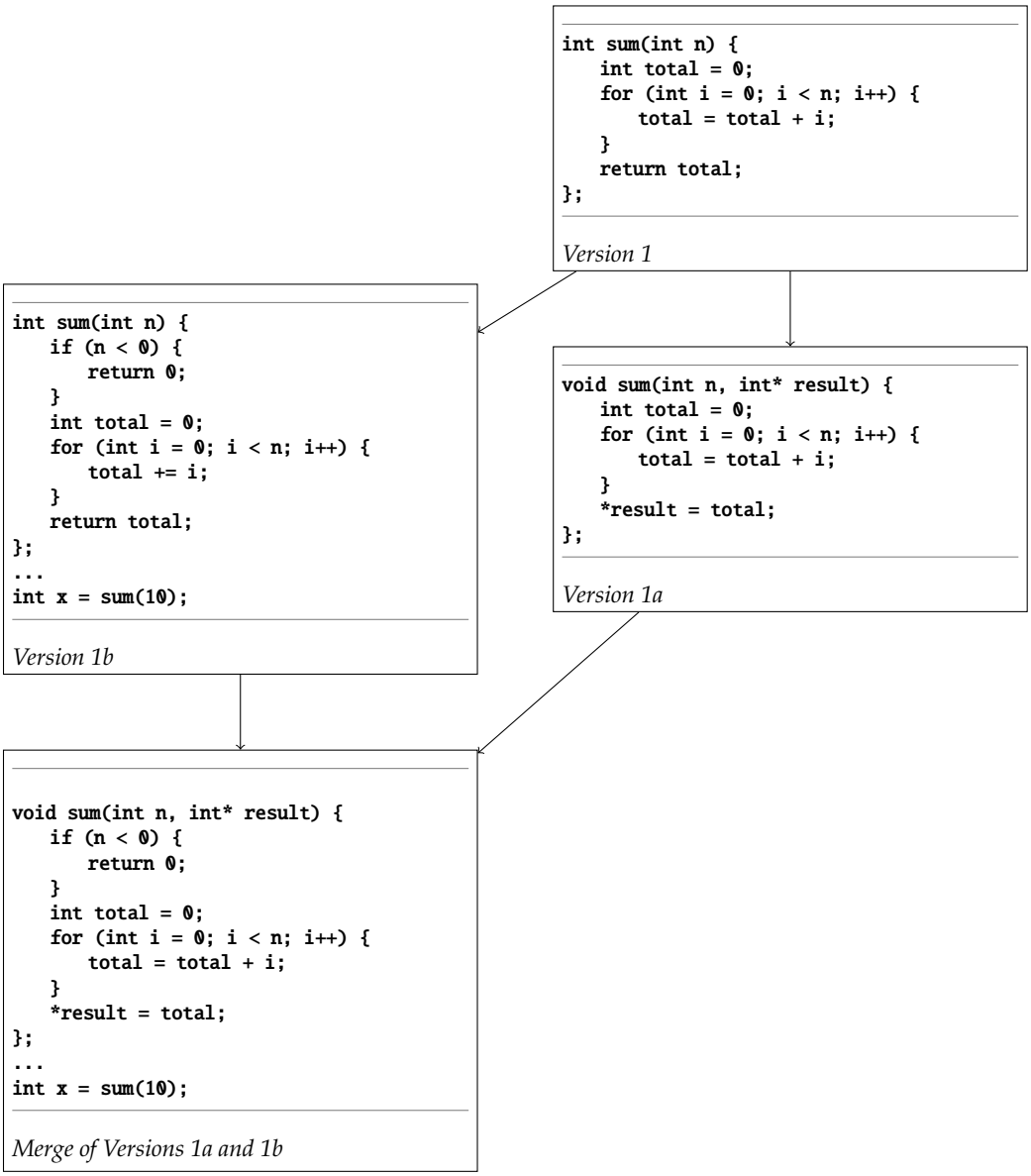


Figure 24.1: An example of a 3-way line based merge.

24.2 Hooks

What are hooks in a version control system? Describe their applications.

Solution: Version control systems such as git and SVN allow users to define actions to take around events such as commits, locking and unlocking files, and altering revision properties. These actions are specified as executable programs known as hook scripts. When the version control system gets to these events, it will check for the existence of a corresponding hook script, and execute it if it exists.

Hook scripts have access to the action as it is taking place, and are passed corresponding command-line arguments. For example, the pre-commit hook is passed the repository path and the transaction ID for the currently executing commit. If a hook script returns a nonzero exit code, the version control system aborts the action, returning the script's standard error output to the user.

The following hook scripts are used most often:

- *pre-commit:* Executed before a change is committed to the repository. Often used to check log messages, to format files, run a test suite, and to perform custom security or policy checking.
- *post-commit:* Executed once the commit has completed. Often used to inform users about a completed commit, for example by sending an email to the team, or update the bug tracking system.

As a rule, hooks should never alter the content of the transaction. At the very least, it can surprise a new developer; in the worst case, the change can result in buggy code.

Scripting languages

Scripting languages, such as AWK, Perl, and Python were originally developed as tools for quick hacks, rapid prototyping, and gluing together other programs. They have evolved into mainstream programming languages. Some of their characteristics are as follows.

- Text strings are the basic (sometimes only) data type.
- Associative arrays are a basic aggregate type.
- Regexps are usually built in.
- Programs are interpreted rather than compiled.
- Declarations are often not required.

They are easy to get started with, and can often do a great deal with very little code.

24.3 Is scripting is more efficient?

Your manager read a study in which a Python program was found to have taken one-tenth the development time then a program written in C++ for the same application, and now demands that you immediately program exclusively in Python. Tell him about five dangers of doing this.

Solution: First, any case study is entirely anecdotal. It's affected by the proficiency of the developers, the nature of the program being written, etc. So no case study alone is sufficient as the basis for a language choice.

Second, switching development environments for any team is an immediate slowdown, since the developers must learn all the new tricks and unlearn all the old ones. If a team wants to make this kind of switch, it needs to have a very long runway.

Third, scripting languages are simply not good for some tasks. For example, many high-performance computing or systems tasks should not be done in Python. The lack of a type system means some serious bugs can go undetected.

Fourth, choosing to program entirely in any language is a risky endeavor. Any decent engineering department should be willing to adopt the best tool for the job at hand.

Finally, Python itself may be a risky choice at this point in time, since there is an ongoing shift from the 2.x branch to 3.x. If you choose to work in 2.x, you likely have a massive refactor into 3.x

coming in the next few years; and if you choose to work in 3.x, you likely will run into libraries you simply cannot use because they don't yet support the latest versions of Python.

24.4 Polymorphism with a scripting language

Briefly, a Java interface is a kind of type specification. To be precise, it specifies a group of related methods with empty bodies. (All methods are nonstatic and public.) Interfaces have many benefits over, for example, abstract classes—in particular, they are a good way to implement polymorphic functions in Java.

Describe the advantages and disadvantages of Python and Java for polymorphism. In particular, how would you implement a polymorphic function in Python?
Solution: Java's polymorphism is first-class. The compiler is aware at compile-time of whatever interfaces (in the general sense) an object exposes—regardless of whether they derive from actual interfaces or from superclasses. As a result, a polymorphic function in Java has a guarantee that any argument to it implements the methods it will call during its execution.

In Python, however, no such safety exists. So there are two tricks to use when implementing a polymorphic function in Python. First, you can specify your classes using multiple inheritance. In this case, many of the superclasses are simply interface specifications. These classes, mix-ins, may implement the desired methods or leave them for the subclass to implement. In effect, they provide a poor-man's interface. (However, they're not as robust as Java's solution since the method resolution order gets tricky in multiple-inheritance scenarios.)

But there is a trickier problem here, which is that even if a class is implemented with a particular method, it's trivial at runtime to overwrite that method, e.g., with `None`, before passing it as an argument. So you still don't have any guarantees. This motivates the second trick, which is the use of "duck typing"—that is, rather than trying to get some safety guarantee that cannot exist in Python, just inspect the object at runtime to see if it exposes the right interface. (The phrase comes from this adage: "If it looks like a duck, quacks like a duck, and acts like a duck, then it's probably a duck".) So, where in Java a developer might write a function that only accepts arguments that implement a `Drawable` interface, in Python you would simply call that object's draw method and hope for the best. Type checking still exists here; it's just deferred to runtime, which implies it may throw exceptions. The function is still quite polymorphic—more so, in some sense, since it can accept literally any object that has a draw method.

Build systems

A program typically cannot be executed immediately after change: intermediate steps are required to build the program and test it before deploying it. Other components can also be affected by changes, such as documentation generated from program text. Build systems automate these steps.

24.5 Dependency analysis

Argue that a build system such as Make which looks purely at the time stamps of source code and derived products when determining when a product needs to be rebuilt can end up spending more time building the product that is needed. How can the you avoid this? Are there any pitfalls to your approach?
Solution: Suppose a source file is changed without its semantics being affected, e.g., a comment is added or it's reformatted. A build system like Make will propagate this change through dependencies, even though subsequent products are unchanged.

One solution to this is as follows. During the reconstruction of a component A, a component called A_{new} is created and compared with the existing A. If A_{new} and A are different, A_{new} is copied to A. Otherwise, A remains unchanged, include the date that it was changed.

This may not always be the reasonable thing to do. For example, even if the source files remain unchanged, environmental differences or changes to the compilation process may lead to the creation of different intermediate targets, which would necessitate a full rebuild. Therefore each intermediate target must be rebuilt in order to compare against the existing components, but it's indeterminate before construction whether the new objects will be used or not.

24.6 ANT vs. Maven

Contrast ANT and Maven

Solution: Historically, ANT was developed in the early days of Java to overcome the limitations of Make. It has tasks as a "primitive"—in Make these must be emulated using pseudo-targets, variables, etc. ANT is cross-platform—since Make largely scripts what you would do at the command-line, a Makefile written for Unix-like platforms will not work out-of-the-box on, say, Windows.

ANT itself has a number of shortcomings. These are overcome in Maven, as described below.

- Maven is declarative whereas Ant is imperative in nature. You tell Maven what are the dependencies of your project—Maven will fetch those dependencies and build the artifact for you, e.g., a war file. In Ant, you have to provide all the dependencies of your project and tell Ant where to copy those dependencies.
- With Maven, you can use "archetypes" to set up your project very quickly. For example you can quickly set up a Java Enterprise Edition project and get to coding in a matter of minutes. Maven knows what should be the project structure for this type of project and it will set it up for you.
- Maven is excellent at dependency management. Maven automatically retrieves dependencies for your project. It will also retrieve any dependencies of dependencies. You never have to worry about providing the dependent libraries. Just declare what library and version you want and Maven will get it for you.
- Maven has the concept of snapshot builds and release builds. During active development your project is built as a snapshot build. When you are ready to deploy the project, Maven will build a release version for you which can be put into a company wide Nexus repository. This way, all your release builds are in one repository that are accessible throughout the company.
- Maven favors convention over configuration. In Ant, you have to declare a bunch of properties before you can get to work. Maven uses sensible defaults and if you follow the Maven recommended project layout, your build file is very small and everything works out of the box.
- Maven has very good support for running automated tests. In fact, Maven will run your tests as part of the build process by default. You do not have to do anything special.
- Maven provides hooks at different phases of the build and you can perform additional tasks as per your needs

Database

Most software systems today interact with databases, and it's important for developers to have basic knowledge of databases.

24.7 SQL vs. NoSQL

Contrast SQL and NoSQL databases.

Solution: A relational database is a set of tables (also known as relations). Each table consists of rows. A row is a set of columns (also known as fields or attributes), which could be of various types (integer, float, date, fixed-size string, variable-size string, etc.). SQL is the language used to create and manipulate such databases. MySQL is a popular relational database.

A NoSQL database provides a mechanism for storage and retrieval of data which is modeled by means other than the tabular relations used in relational databases. MongoDB is a popular NoSQL database. The analog of a table in MongoDB is a collection, which consists of documents. Collections do not enforce a schema. Documents can be viewed as hash maps from fields to values. Documents within a collection can have different fields.

A key benefit of NoSQL databases is simplicity of design: a field can trivially be added to new documents in a collection without this affecting documents already in the database. This makes NoSQL a popular choice for startups which have an agile development environment.

The data structures used by NoSQL databases, e.g., key-value pairs in MongoDB, are different from those used by default in relational databases, making some operations faster in NoSQL. NoSQL databases are typically much simpler to scale horizontally, i.e., to spread documents from a collection across a cluster of machines.

A key benefit of relational databases include support for ACID transactions. ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. For example, there is no way in MongoDB to remove an item from inventory and add it to a customer's order as an atomic operation.

Another benefit of relational database is that their query language, SQL, is declarative, and relatively simple. In contrast, complex queries in a NoSQL database have to be implemented programmatically. (It's often the case that the NoSQL database will have some support for translating SQL into the equivalent NoSQL program.)

24.8 Normalization

What is database normalization? What are its advantages and disadvantages?

Solution: Database normalization is the process of organizing the columns and tables of a relational database to minimize data redundancy. Specifically, normalization involves decomposing a table into less redundant tables without losing information, thereby enforcing integrity and saving space.

The central idea is the use of "foreign keys". A foreign key is a field (or collection of fields) in one table that uniquely identifies a row of another table. In simpler words, the foreign key is defined in a second table, but it refers to the unique key in the first table. For example, a table called Employee may use a unique key called employee_id. Another table called Employee Details has a foreign key which references employee_id in order to uniquely identify the relationship between both the tables. The objective is to isolate data so that additions, deletions, and modifications of an attribute can be made in just one table and then propagated through the rest of the database using the defined foreign keys.

The primary drawback of normalization is performance—often, a large number of joins are required to recover the records the application needs to function.

Write SQL commands to create a table appropriate for representing students as well as SQL commands that illustrate how to add, delete, and update students, as well as search for students by GPA with results sorted by GPA.

Then add a table of faculty advisors, and describe how to model adding an advisor to each student. Write a SQL query that returns the students who have an advisor with a specific name.

Solution: Natural fields for a student are name, birthday, GPA, and graduating year. In addition, we would like a unique key to serve as an identifier. A SQL statement to create a student table would look like the following: `CREATE TABLE students (id INT AUTO_INCREMENT, name VARCHAR(30), birthday DATE, gpa FLOAT, graduate_year INT, PRIMARY KEY(id));`. Here are examples of SQL statements for updating the students table.

- Add a student
  - `INSERT INTO students(name, birthday, gpa, graduate_year) VALUES ('Cantor', '1987-10-22', 3.9, 2009);`
- Delete students who graduated before 1985.
  - `DELETE FROM students WHERE graduate_year < 1985;`
- Update the GPA and graduating year of the student with id 28 to 3.14 and 2015, respectively.
  - `UPDATE students SET gpa = 3.14, graduate_year = 2015 WHERE id = 28;`

The following query returns students with a GPA greater than 3.5, sorted by GPA. It includes each student's name, GPA, and graduating year. `SELECT gpa, name, graduate_year FROM students WHERE gpa > 3.5 ORDER BY gpa DESC;`

Now we turn to the second part of the problem. Here is a sample table for faculty advisers.

id	*name*	*title*
1	Church	Dean
2	Tarski	Professor

We add a new column advisor_id to the students table. This is the foreign key on the preceding page. As an example, here's how to return students with GPA whose advisor is Church: `SELECT s.name, s.gpa FROM students s, advisors p WHERE s.advisor_id = p.id AND p.name = 'Church';`.

Variant: What is a SQL join? Suppose you have a table of courses and a table of students. How might a SQL join arise naturally in such a database?

Networking

Most software systems today interact with the network, and it's important for developers even higher up the stack to have basic knowledge of networking. Here are networking questions you may encounter in an interview setting.

24.10 IP, TCP, AND HTTP

Explain what IP, TCP, and HTTP are. Emphasize the differences between them.

Solution: IP, TCP, and HTTP are networking protocols—they help move information between two hosts on a network.

IP, the Internet Protocol, is the lowest-level protocol of the three. It is independent of the physical channel (unlike for example, WiFi or wired Ethernet). It's primary focus is on getting individual packets from one host to another. Each IP packet has a header that include the source

and destination IP address, the length of the packet, the type of the protocol it's layering (TCP and UDP are common), and an error checking code.

IP moves packets through a network consisting of links and routers. A router has multiple input and output ports. It forwards packets through the network using routing tables that tell the router which output port to forward in incoming packet to. Specifically, a routing table maps IP prefixes to output ports. For example, an entry in the routing table may indicate that all packets with destination IP address matching `171.23.*.*` are to be forwarded out the interface port with id 17. IP routers exchange "link state information" to compute the routing tables; this computation is a kind of shortest path algorithm.

TCP, the Transmission Control Protocol, is an end-to-end protocol built on top of IP. It creates a a continuous reliable bidirectional connection. It does this by maintaining state at the two hosts—this state includes IP packets that have been received, as well as sending acknowledgements. It responds to dropped packets by requesting retransmits—each TCP packet has a sequence number. The receiver signals the transmitter to reduce its transmission rate if the drop rate becomes high—it does this by reducing the received window size. TCP also "demultiplexes" incoming data using the concept of port number—this allows multiple applications on a host with a single IP address to work simultaneously.

HTTP, the Hyper Text Transfer Protocol, is built on top of TCP. An HTTP request or response can be viewed as a text string consisting of a header and a body. The HTTP request and responses header specify the type of the data through the Content-Type field. The response header includes a code, which could indicate success ("200"), a variety of failures ("404"—resource not found), or more specialized scenarios ("302"—use cached version). A few other protocol fields are cookies, request size, compression used.

HTTP was designed to enable the world-wide web, and focused on delivering web pages in a single logical transaction. It's generality means that it has grown to provide much more, e.g., it's commonly used to access remote procedures ("services"). An object—which could be a web page, a file, or a service—are addressed using a URL, Uniform Resource Locator, which is a domain name, a path, and optional arguments. When used to implement services, a common paradigm is for the client to send a request using URL arguments, a JSON object, and files to be uploaded, and the server responds with JSON.

24.11 HTTPS

Describe HTTPS operationally, as well as the ideas underlying it.

Solution: In a nutshell, HTTPS is HTTP with SSL encryption. Because of the commercial importance of the Internet, and the prevalence of wireless networks, it's imperative to protect data from eavesdroppers—this is not done automatically by IP/TCP/HTTP.

Conceptually, one way to encrypt a channel is for the communicating parties to XOR each transmitted byte with a secret byte—the receiver then XORs the received bytes with the secret. This scheme is staggeringly weak—an eavesdropper can look at statistics of the transmitted data to easily figure out which of the 256 bytes is the secret. It leaves open the question of how to get the secrete byte to the receiver. Furthermore, a client who initiates a transaction has no way of knowing if someone has hijacked the network and directed traffic from the client to a malicious server.

SSL resolves these issues. First of all, it uses public key cryptography for key exchange—the idea is that the public key is used by the client to encrypt and send a secret key (which can be viewed as a generalization of the secret byte; 256 bits is the common size) to the server. This secret key is used to scramble the data to be transmitted in a recoverable way. The secret key is generated via

hashing a random number. Public key cryptography is based on the fact that it's possible to create a one-way function, i.e., a function that can be made public and is easy to compute in one direction, but hard to invert with the public information about it. Rivest, Shamir, and Adleman introduced a prototypical such function which is based on several number theoretic facts, such as the relative ease with which primes can be created, that computing $a^b \mod c$ is fast, and that factorization is (likely) hard.

Private key cryptography involves a single key which is a shared secret between the sender and recipient. Private key cryptography offers many orders of magnitude higher bandwidth than public key cryptography. For this reason, once the secret key has been exchanged via public key cryptography, HTTPS turns to private key cryptography, specifically AES. AES operates on 128 bit blocks of data. (If the data to be transmitted is less than 128 bits, it's padded up; if it's greater than 128 bits, it's broken down into 128 bit chunks.) The 128 bit block is represented as 16 bytes in a 4×4 matrix. The secret key is used to select an ordering in which to permute the bytes; it is also used to XOR the bits. This operation is reversible using the secret key.

A small shortcoming of the scheme described above is that it is susceptible to replay attacks—an adversary may not know the exact contents of a transmission, but may know that it represent an action ("add 10 gold coins to my World-Of-Warcraft account"). This can be handled by having the server send a random number which must accompany client requests.

HTTPS adds another layer of security—SSL certificates. The third party issuing the SSL certificate verifies that the HTTPS server is on the domain that it claims to be on, which precludes someone who has hijacked the network from pretending to be the server being requested by the client. This check is orthogonal to the essence of SSL (private key exchange and AES), and it can be disabled, e.g., for testing a server locally.

24.12 DNS

Describe DNS, including both operational as well as implementation aspects.

Solution: DNS is the system which translates domain names—`www.google.com`—to numerical IP addresses—`216.58.218.110`. It is essential to the functioning of the Internet. DNS is what makes it possible for a browser to access a website—the routers in the Internet network rely on IP addresses to lookup next-hops for packets.

There are many challenges to implementing DNS. Here are some of the most important ones.

- There are hundreds of millions of domains.
- There are hundreds of billions of DNS lookups a day.
- Domains and IP addresses are added and updated continuously.

To solve these problems, DNS is implemented as a distributed directory service. A DNS lookup is addressed to a DNS server. Each DNS server stores a database of domain names to IP addresses; if it cannot find a domain name being queried in this database it forwards the request to other DNS servers. These requests can get forwarded all the way up to the root name servers, which are responsible for top-level domains.

At the time of writing there are 13 root name servers. This does not mean there are exactly 13 physical servers. Each operator uses redundancy to provide reliable service. Additionally, these servers are replicated across multiple locations. Key to performance is the heavy use of caching, in the client itself, as well as in the DNS servers.