CSCI2100 Data Structures

Ryan Chan

May 5, 2025

Abstract

This is a note for **CSCI2100 Data Structures**.
Contents are adapted from the lecture notes of CSCI2100, prepared by Irwin King, as well as some online resources.
This note is intended solely as a study aid. While I have done my best to ensure the accuracy of the content, I do not take responsibility for any errors or inaccuracies that may be present. Please use the material thoughtfully and at your own discretion.
If you believe any part of this content infringes on copyright, feel free to contact me, and I will address it promptly.
Mistakes might be found. So please feel free to point out any mistakes.

Contents

Chapter 1

Introduction

1.1 Overview

A **data structure** is a way to organize and store data in a computer program, allowing for efficient access and manipulation.

An **algorithm** is different from a **program**. An algorithm is a process or set of rules used for calculation or problem-solving. It is a step-by-step outline or flowchart showing how to solve a problem. A program, on the other hand, is a series of coded instructions that control the operation of a computer or other machines. It is the implemented code of an algorithm.

For example, to solve the greatest common divisor (GCD) problem, we can use the following algorithm.

Algorithm 1.1: Euclid's Algorithm

```
Data: $m, n \in \mathbb{Z}^+$
Result: GCD($m, n$)
while $m > 0$ do
    if $n > m$ then
        swap $m$ and $n$
    subtract $n$ from $m$
return $n$
```

By using a mathematical method to prove this algorithm, we can show that it is correct, provided that it terminates.

Having proved the correctness, we also need to use different test cases to check if there is anything wrong with the coding or the proof. We should consider special cases, including large values, swapped values, etc.

We are also interested in the time and space (computer memory) it uses, which we call **time complexity** and **space complexity**. Typically, complexity is a function of the values of the inputs, and we would like to know which function. We can also consider the best case, average case, and worst-case scenarios.

For example, in the above algorithm, the best case would be $m = n$, with just one iteration. If $n = 1$, there are m iterations, which is the worst case. However, for the average case, it is difficult to analyze.

Also, for space complexity, it is constant since we only use space for the three integers: m, n, and t.

To improve the above algorithm, we can use `mod`, so we don't need to keep doing subtraction.

1.2 Algorithm

An algorithm is a finite set of instructions which, if followed, accomplishes a particular task. Every algorithm must satisfy the following criteria:

- Input: There are zero or more quantities that are externally supplied.

- Output: At least one quantity is produced.

- Definiteness: Each instruction must be clear and unambiguous.

- Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm will terminate after a finite number of steps.

- Effectiveness: Every instruction must be sufficiently basic that it can, in principle, be carried out by a person using only pencil and paper.

We also define an algorithm as any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. It is thus a sequence of computational steps that transform the input into output.

It can also be viewed as a tool for solving a well-specified computational problem. The problem statement specifies, in general terms, the desired input or output relationship, and the algorithm describes a specific computational procedure for achieving that input or output relationship.

An algorithm is said to be correct if, for every input instance, it halts with the correct output. It can solve the given computational problem. In contrast, an incorrect algorithm might not halt at all on some input instances, and sometimes it can even produce useful results.

1.3 Study of Data

A data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

A data structure is a set of domains D, a designated domain $d \in D$, a set of functions F, and a set of axioms A.

An implementation of a data structure d is a mapping from d to a set of other data structures e.

Chapter 2

Analysis

2.1 Complexity

Before, we talked about the definition of an algorithm. In this part, we would like to know how we can estimate the time required for a program, how to reduce the running time of a program, what the storage complexity is, and how to deal with trade-offs.

We can analyze the runtime by comparing functions. For example, given two functions $f(N)$ and $g(N)$, we can compare their relative rates of growth. There are three types of comparisons that we can make: $f(n) = \Theta(g(n))$ represents the exact bound, $f(n) = O(g(n))$ represents the upper bound, and $f(n) = \Omega(g(n))$ represents the lower bound.

By using bounds, we can establish a relative order among functions. Here, we often use $O(n)$ to analyze time complexity.

For the definition of the upper bound, it says that there is some point n_0 past which $cf(N)$ is always at least as large as $T(N)$. Then we say that $T(N) = O(f(N))$, where $f(N)$ is the upper bound on $T(N)$.

Definition 2.1.1. We say that $f(n) = O(g(n))$ iff there exists a constant $c > 0$ and an $n_0 \geq 0$ such that

$$f(n) \leq cg(n) \quad \text{for all } n \geq n_0$$

Or we can use the following notation

$$\exists c > 0, n_0 \geq 0 \text{ such that } f(n) \leq cg(n) \forall n \geq n_0$$

There are some rules to follow:

- Transitivity

$$\text{If } f(n) = O(g(n)) \text{ and } g(n) = O(h(n)), \text{ then } f(n) = O(h(n))$$

- Rule of sums

$$f(n) + g(n) = O(max\{f(n), g(n)\})$$

- Rule of products

$$\text{If } f_1(n) = O(g_1(n)), f_2(n) = O(g_2(n)), \text{ then } f_1(n)f_2(n) = O(g_1(n)g_2(n))$$

We do not include constants or lower-order terms inside Big-O notation.

If $f(n)$ is a polynomial in n with degree r, then $f(n) = O(n^r)$, but for $s < r$, $f(n) \neq O(n^s)$.

Also, any logarithm of n grows more slowly than any positive power of n as it increases. Hence, $\log n$ is $O(n^k)$ for any $k > 0$, but n^k is never $O(\log n)$ for any $k > 0$.

Order	Time
$O(1)$	constant time
$O(n)$	linear time
$O(n^2)$	quadratic time
$O(n^3)$	cubic time
$O(2^n)$	exponential time
$O(\log n)$	logarithmic time
$O(\log^2 n)$	log-squared time

On a list of length n, sequential search has a running time of $O(n)$.

On an ordered list of length n, binary search has a running time of $O(\log n)$.

The sum of the sums of integer indices of a loop from 1 to n is $O(n^2)$.

In summary, Big-O notation provides an upper bound of the complexity in the **worst-case**, helping to quantify performance as the input size becomes arbitrarily large. However, it doesn't measure the actual time, but represents the number of operations an algorithm will execute.

2.2 Recurrence Relations

Recurrence relations are useful in certain counting problems, for example, recursive algorithms. They relate the n-th element of a sequence to its predecessors.

By definition, a recurrence relation for the sequence $a_0, a_1, \cdots$ is an equation that relates a_n to certain of its predecessors $a_0, a_1, \cdots, a_{n-1}$. Initial conditions for the sequence are explicitly given values for a finite number of the terms of the sequence.

To solve a recurrence relation, we can use iteration. We use the recurrence relation to write the n-th term a_n in terms of certain of its predecessors. We then successively use the recurrence relation to replace each of $a_{n-1}, \cdots$ by certain of their predecessors. We continue until an explicit formula is obtained.

For example, the Fibonacci sequence is also defined by the recurrence relation.

Example (Tower of Hanoi)**.** Find an explicit formula for a_n, the minimum number of moves in which the n-disk Tower of Hanoi puzzle can be solved.

Given $a_n = 2a_{n-1} + 1$, $a_1 = 1$, by applying the iterative method, we obtain:

$$\begin{aligned}
a_n &= 2a_{n-1} + 1 \\
&= 2(2a_{n-2} + 1) + 1 \\
&= 2^2 a_{n-2} + 2 + 1 \\
&= 2^2(2a_{n-3} + 1) + 2 + 1 \\
&= 2^3 a_{n-3} + 2^2 + 2 + 1 \\
&= \cdots \\
&= 2^{n-1} a_1 + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1 \\
&= 2^n - 1
\end{aligned}$$

Chapter 3

ADT, List, Stack and Queue

3.1 Abstract Data Type (ADT)

We use data abstraction to simplify software development since it facilitates the decomposition of the complex task of developing a software system. To put it simply, data abstraction shows only the essential details of data, while the implementation details are hidden.

For example, as will be discussed later, List, Stack, and Queue (LSQ) are forms of data abstraction, or what we call abstract data types. We can use them to retrieve or store data, but we don't know how they are actually stored or indexed.

Data encapsulation, or information hiding, is the concealing of the implementation of a data object from the outside world. Through data abstraction, we can separate the specification of a data object from its implementation.

A data type is a collection of objects and a set of operations that act on those objects. An abstract data type (ADT) is a data type organized in such a way that we can separate the specification of the object and the specification of the operations on the object. Abstract data types are simply a set of operations, and they are mathematical abstractions.

Note that abstraction is like a functional description without knowing how to use it, while implementation, on the contrary, is something that can be used and executed.

In summary, an ADT is a high-level description of how data is organized and the operations that can be performed on it. It abstracts the details of its implementation and only exposes the operations that are allowed on data structures.

3.2 List

The first abstract data type in this chapter is List.

3.2.1 Definition

When dealing with a general list of the form $a_1, a_2, \cdots, a_n$, we say that the size of this list is n. If the list is of size 0, we call it the **null list**. Except null list, we say that a_{i+1} follows/succeeds $a_i (i < n)$ and that a_{i-1} precedes $a_i (i > 1)$.

The first element of the list is a_1, and the last element is a_n. The predecessor of a_1 and the successor of a_n is not defined.

3.2.2 Operations

A list of elements of type T is a finite sequence of elements of T together with the following operations:

- Create the list and make it empty.
- Determine whether the list is empty or not.
- Determine whether the list is full or not.
- Find the size of the list.
- Retrieve any entry from the list, provided that the list is not empty.
- Store a new entry, replacing the entry at any position in the list, provided that the list is not empty.
- Insert a new entry into the list at any position, provided that the list is not full.
- Delete any entry from the list, provided that the list is not empty.
- Clear the list to make it empty.

With these operations, we can perform various tasks on the list ADT.

3.2.3 Implementation

We can use an array to implement a list. Most of the operations follow linear time, for example, `print_list`, `make_null`, `find`, and `find_kth`. For insertion, there could be cases where the list is full, and that's why we need dynamically allocated space. For deletion, we need to find the element, perform the deletion, and reallocate space, which might require more time. Thus, we introduce the linked list.

3.2.4 Linked List

There are several types of linked lists:

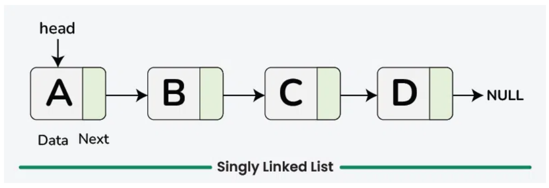


Figure 3.1: Singly Linked List

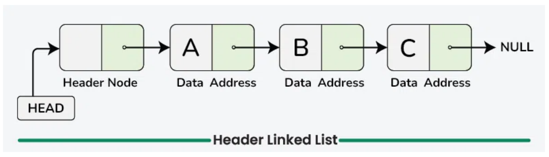


Figure 3.2: Singly Linked List with Header

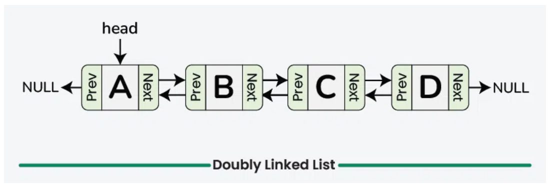


Figure 3.3: Doubly Linked List

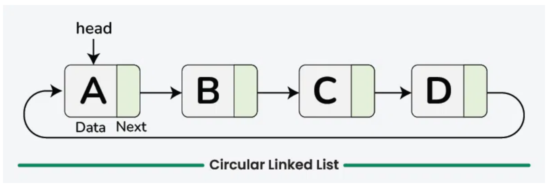


Figure 3.4: Circularly Linked List

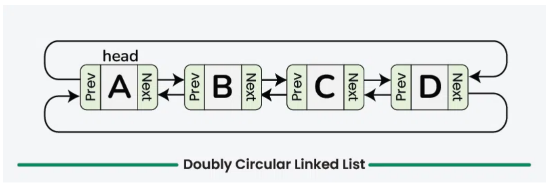


Figure 3.5: Circularly Doubly Linked List

A polynomial can be represented as

$$F(X) = \sum_{i=0}^{N} A_i X^i$$

For example, $F(X) = 4X^3+2X^2+5X+1$. We may want to perform operations like addition, subtraction, multiplication, and differentiation. Using an array data structure, the time complexity may be larger due to the need to store all terms, including zero coefficients. However, with a linked list, we can efficiently perform these operations by traversing the linked list and processing only the non-zero terms.

Also, note that a circular list saves space but not time. It is useful for smaller datasets. However, for a larger number of students and courses, the use of such a circular list might be a waste of space.

In summary, a list abstract data type represents an ordered collection of elements. They can be added or removed at any position in the list. It provides methods to access elements by their position.

By using different types of linked lists, we can achieve various goals. For example, we can print all the elements in reverse using a doubly linked list.

3.3 Stack

3.3.1 Definition

A stack is an ordered list in which all insertions and deletions are made at one end, called the top. It follows the Last In, First Out (LIFO) rule.

3.3.2 Operations

A stack of elements of type T is a finite sequence of elements of T along with the following operations:

- Create the stack.
- Determine if the stack is empty or not.
- Determine if the stack is full or not.
- Determine the number of entries in the stack.
- Insert (Push) a new entry at one end of the stack, called its top, if the stack is not full.
- Retrieve the entry at the top of the stack, if the stack is not empty.
- Delete (Pop) the entry at the top of the stack, if the stack is not empty.
- Clear the stack to make it empty.

3.3.3 Implementation

We can use a doubly linked list to implement the stack, where both Push and Pop operations happen at the front of the list. We can use the Top operation to examine the element at the front of the list. In this case, the space complexity would be $O(3n)$, and the time complexity would be $O(c)$, where c is a constant.

Alternatively, we can use an array to implement the stack, since we only perform insertion and deletion at the top. However, we need to declare the size ahead of time and use TopOfStack as the counter to point to the top of the stack. If an array is used, the space complexity would be $O(n)$, and the time complexity would be $O(1)$.

3.3.4 Application

Balance Symbols

We can use a stack to balance symbols, which is commonly used in compilers to check for syntax errors. While compiling, an empty stack is created, and an opening symbol is pushed onto the stack. Then, when a closing symbol is encountered, it is popped from the stack. There could be four types of errors:

1. Stack Overflow: Too many brackets.

2. Mismatched Symbols: The opening and closing symbols don't match.
3. Empty Stack: Attempting to pop from an empty stack.
4. Non-Empty Stack: The stack isn't empty at the end of the process.

Reverse Polish Calculator

We can also use a stack to make a Reverse Polish Calculator. There are three forms of notation: prefix, postfix, and infix. For example, if the expression is $a \times b$, then:

1. Prefix: $\times ab$
2. Postfix: $ab\times$
3. Infix: $a \times b$

In Reverse Polish notation (postfix), parentheses are not needed. Using a stack, we can calculate the answer by evaluating the postfix expression. For example, when the character is a number, it is pushed onto the stack. If the character is an operator, two elements are popped from the stack, and the operation is performed on those two elements.

In summary, the stack abstract data type is a collection of elements with two main operations: push and pop. All operations happen at the top, and it follows the Last In, First Out (LIFO) approach.

3.4 Queue

3.4.1 Definition

A Queue is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. It follows the First In, First Out (FIFO) rule.

3.4.2 Operations

Several operations can be performed on a queue:

- Create the queue
- Determine if the queue is empty or not.
- Insert (Enqueue) a new entry at one end of the queue, called its rear, if the queue is not full.
- Delete (Dequeue) an entry at the other end of the queue, called its front, if the queue is not empty.
- Retrieve the entry at the front of the queue, if the queue is not empty.
- Clear the queue and make it empty.

3.4.3 Implementation

We can use both linear and circular arrays to implement a queue. For a linear array, we can have two indices that always increase. However, this might lead to overflow. Additionally, the array may need to be shifted forward or backward after each enqueue or dequeue operation. For a circular array, we have the following possibilities:

- Front and rear indices, with one position left vacant.
- Front and rear indices, with a Boolean variable indicating fullness or emptiness.
- Front and rear indices, with an integer variable counting entries.
- Front and rear indices taking special values to indicate emptiness.

3.4.4 Application

Queues are commonly used in various applications, such as in printer queues, airline control systems, and bank queues.

In summary, the physical model of a queue is a linear array, with the front always in the first position. All entries are moved up the array whenever the front is deleted. It follows the First In, First Out (FIFO) rule.

Chapter 4

Trees

In this chapter, we will introduce some fundamental concepts of trees. Trees are hierarchical data structures widely used for various applications such as representing hierarchical relationships, optimizing search operations, and organizing data efficiently.

4.1 General Tree

4.1.1 Nodes

A tree is a collection of nodes. The collection can be empty, which is sometimes denoted as A. Otherwise, a tree consists of a distinguished node r, called the root, and zero or more subtrees $T_1, T_2, \cdots, T_k$, each of whose roots are connected by a directed edge to r. The root of each subtree is said to be a child of r, and r is the parent of each subtree root.

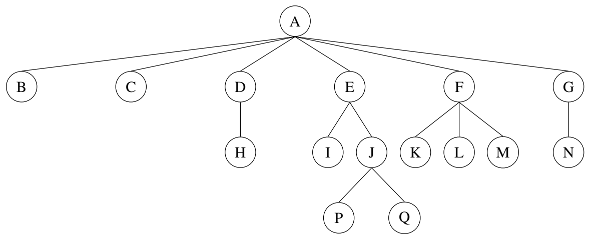


Figure 4.1: General Tree

Each node in a tree has a parent and may have an arbitrary number of children, possibly zero. Nodes with no children are known as leaves, and nodes with the same parent are called siblings. For example, in the above graph, A is the parent of D, and B, C, and D are siblings.

A path from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$. The length of this path is the number of edges on the path, namely $k - 1$. There is a path of length zero from every node to itself, and there is exactly one path from the root to each node.

Also, if there is a path from n_1 to n_2, we call n_1 the ancestor of n_2, while n_2 is the descendant of n_1. If $n_1 \neq n_2$, we call them a proper ancestor or proper descendant.

For any node n_i, the depth of n_i is the length of the unique path from the root to n_i. Thus, the root is at depth 0. The height of n_i is the longest path from n_i to a leaf. Therefore, all leaves are at height 0. For example, in the graph above, E is at depth 1 and height 2.

The height of a tree is equal to the height of the root, and the depth of a tree is the depth of the deepest leaf. These two values are always equal, representing the longest path from the root to any leaf.

To implement a tree, we could store both the data and a pointer to each child in the node. However, this approach might not work well for a large number of children. We can solve this issue by keeping the children of each node in a linked list of tree nodes.

4.1.2 Traversal

Suppose we have a directory that includes files and subdirectories. How do we list the names of all the files?

We can use a technique called tree traversal. To traverse a data structure means to process every node in the data structure exactly once, in whatever way you choose. It is possible to pass the same node multiple times, but it would only be processed once.

There are three main traversal orders: preorder, inorder, and postorder traversal. However, as long as the traversal follows a systematic way of processing data, it is valid.

For example, in the graph on the right, we have:

- Preorder: 013425786
- Inorder: 314075826
- Postorder: 341785620
- Level-order: 012345678

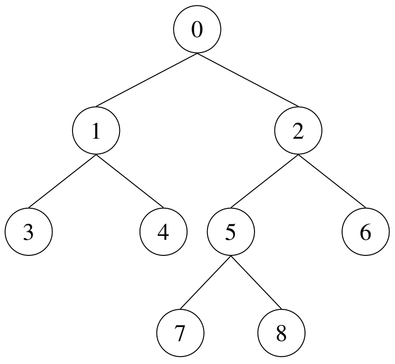


Figure 4.2: Traversal Demonstration

4.2 Binary Tree

4.2.1 Definition

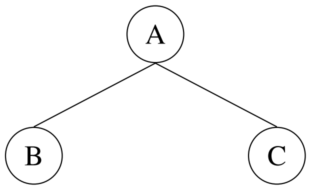


Figure 4.3: Generic Binary Tree

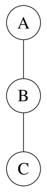


Figure 4.4: Degenerated Binary Tree

A binary tree is a tree in which no node has more than two children. The depth of an average binary tree is considerably smaller than n. For example, even in the worst case — the degenerate tree — the depth would be $n - 1$. The average depth is $O(\frac{h}{2})$, and for a special type of binary tree, namely the binary search tree, the average depth is $O(\log n)$.

4.2.2 Some Binary Trees

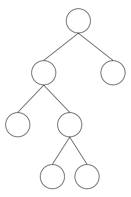


Figure 4.5: Full Binary Tree

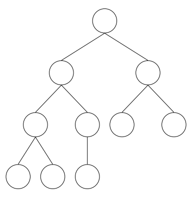


Figure 4.6: Complete Binary Tree

A full binary tree is a binary tree in which every node, other than the leaves, has exactly two children.

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all the nodes in the last level are as far left as possible. This means the nodes in the last level are filled from left to right.

4.2.3 Implementation

Since a binary tree has at most two children, we can implement it by keeping direct pointers to them. A node can be represented as an element in a doubly linked list, storing key information along with two pointers to its left and right children.

We can also implement a node using two pointers: one pointing to its left child and the other pointing to its next sibling.

4.3 Expression Tree

An expression tree is also a binary tree, which is used to calculate the result of an expression. For example, we can express the expression $((9 - (2 + 3)) * (7 - 1))$ using the expression tree on the right.

In an expression tree, the leaves represent operands, and the internal nodes represent operators. By using inorder traversal, we can recover the original expression. From the binary tree, using postorder traversal, we can obtain the postfix notation. With the use of a stack, we can implement a calculator, as shown in the previous chapter.

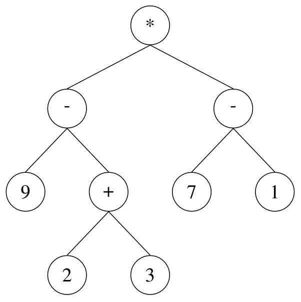


Figure 4.7: Expression Tree

4.4 Binary Search Tree

4.4.1 Definition

A binary search tree (BST) has the same physical property as a binary tree, meaning that nodes have at most two children. However, it also has an ordering property: for each node, all the nodes in its left subtree have smaller values, and all the nodes in its right subtree have larger values. This ordering property turns a binary tree into a binary search tree, and it implies that all the elements in the tree can be ordered in a consistent manner.

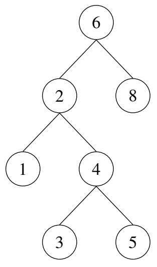


Figure 4.8: Binary Search Tree

4.4.2 Operations

There are some typical operations that can be done on a binary search tree, like make null, find, find max, find min, insertion and deletion.

For the `Find` operation, it generally requires returning a pointer to the node in tree T that has key x, or `null` if there is no such node. The structure of the tree makes this simple. If T is empty, then we can just return `null`. If the key stored at T is x, we can return T. Otherwise, we make a recursive call on a subtree of T, either left or right, depending on the relationship of x to the key stored in T.

For the `Find_min` and `Find_max` operations, these routines return the position of the smallest and largest elements in the tree, respectively. To perform `Find_min`, start at the root and go left as long as there is a left child. The stopping point is the smallest element. The `Find_max` routine is the same, except that branching is to the right child.

For insertion, we proceed down the tree. If x is found, we do nothing (or "update" something). Otherwise, we insert x at the last spot on the path traversed. Duplicates can be handled by keeping an extra field in the node record that indicates the frequency of occurrence.

However, for deletion, it is more difficult since we need to consider several possibilities. If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node. The complicated case is when we need to delete a node with two children. The general idea is to replace the key of the node with the smallest (leftmost) key of the right subtree and recursively delete the node.

4.4.3 Analysis

As mentioned before, the average depth of a binary search tree is $O(\log n)$. Therefore, intuitively, all operations, except `make_null`, should take $O(\log n)$ time. The running time of all the operations, except `make_null`, is $O(d)$, where d is the depth of the node containing the accessed key. However, how do we get the average depth of $O(\log n)$?

Proof. Let $D(n)$ be the internal path length for some tree T of n nodes. The internal path length is the sum of the depths of all nodes in a tree, and $D(1) = 0$. A n-node tree consists of an i-node left subtree and an $(n-i-1)$-node right subtree, plus a root at depth zero for $0 \le i < n$.

Then we have $D(i)$ as the internal path length of the left subtree with respect to its root, and we obtain

$$D(n) = D(i) + D(n-i-1) + n - 1$$

If all subtree sizes are equally likely, which is true for a binary search tree, then the average value of both $D(i)$ and $D(n-i-1)$ is

$$\frac{1}{n}\sum_{j=0}^{n-1} D(j)$$

Which yields

$$D(n) = \frac{2}{n}\left[\sum_{j=0}^{n-1} D(j)\right] + n - 1$$

■

This recurrence gives an average value of $D(n) = O(n \log n)$. Thus, the expected depth of any node should be $O(\log n)$.

Another thing to notice is that for the deletion operation, it favors the left subtree. So, after many insertions and deletions, we may end up with an unbalanced binary tree, which would look like a degenerated tree. To ensure that all the nodes can be operated on in $O(\log n)$ time, we need to make the binary search tree balanced. This is why we have the AVL tree.

4.5 AVL Tree

4.5.1 Definition

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balancing condition. It is identical to a binary search tree, having the same physical and ordering properties. However, for every node in the AVL tree, the heights of the left and right subtrees can differ by at most 1.

With an AVL tree, all tree operations, except insertion, can be performed in $O(\log n)$ time.

To construct the smallest AVL tree of height n, we can use the two smallest AVL subtrees of heights $n-1$ and $n-2$. By doing so recursively, we can find the smallest AVL tree.

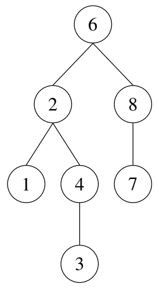


Figure 4.9: AVL Tree

The height of an empty tree is defined to be -1. Height information is kept for each node. The height of an AVL tree is at most roughly $1.44\log(n+2) - 0.328$, but in practice, it is about $\log(n+1) + 0.25$.

4.5.2 Operations

All tree operations can be performed in $O(\log n)$ time, except possibly insertion. Insertion and deletion operations need to update the balancing information since they might violate the AVL tree property. Therefore, we need to restore the property by means of rotations.

A single rotation involves only a few pointer changes and alters the structure of the tree while preserving the search tree property. Rotations happen from the bottom up, meaning we start checking balancing conditions from the lowest affected node and move upward.

For insertion into the right subtree of the right child, we perform a Left Rotation; for insertion into the left subtree of the left child, we perform a Right Rotation.

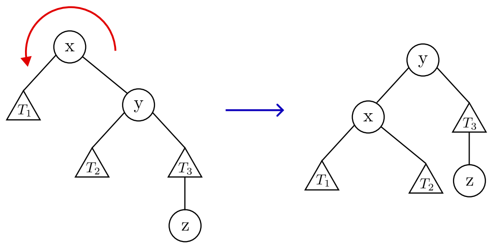


Figure 4.10: Left Rotation

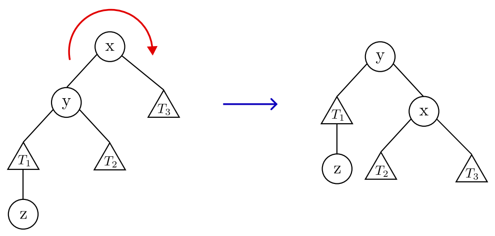


Figure 4.11: Right Rotation

However, single rotation might not work for more complex cases, where the height imbalance is caused by a node inserted into the tree containing the middle element, while the other subtrees have identical heights. In such cases, we use a double rotation, which involves four subtrees instead of three.

For insertion into the right subtree of the left child, we use a Left-Right Rotation. For insertion into the left subtree of the right child, we use a Right-Left Rotation. These double rotations help restore balance by first performing a single rotation on the child node and then performing a second rotation on the parent node, ensuring that the AVL tree's balance property is maintained.

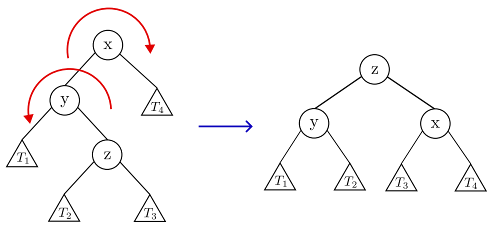


Figure 4.12: Left-Right Rotation

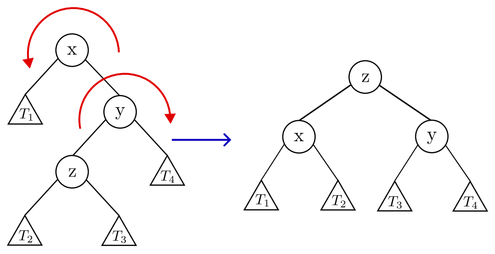
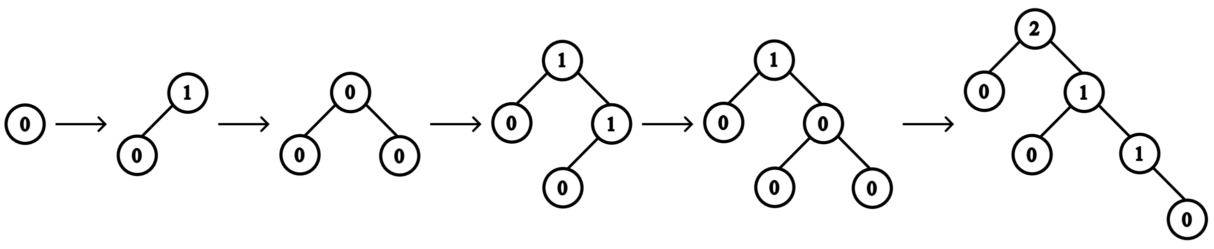


Figure 4.13: Right-Left Rotation

These four rotations help us perform all the necessary balancing operations.

To insert a new node with key x into an AVL tree T, we recursively insert x into the appropriate subtree of T_{lr}. If the height of T_{lr} does not change, then we are done. Otherwise, if a height imbalance occurs, we perform the appropriate single or double rotation depending on x and the keys in T and T_{lr}. Finally, we update the height. For example, the method for checking imbalance is demonstrated below:



The implementation of an AVL tree is similar to a binary search tree, with the addition of height information stored in each node. This height information allows the tree to maintain balance by ensuring that the difference in height between the left and right subtrees of any node is at most 1.

4.6 B-Tree

4.6.1 Definition

A B-Tree of order m is a tree with the following properties:

- All the leaf nodes must be at the same level (have the same depth).

- All non-leaf nodes except the root must have at least $\lceil \frac{m}{2} \rceil - 1$ keys and a maximum of $m - 1$ keys.

- All non-leaf nodes except the root (i.e., all internal nodes) must have children between $\lceil \frac{m}{2} \rceil$ and m.

- The root is either a leaf or has between 2 and m children.

- A non-leaf node with $n - 1$ keys must have n children.

- All the key values within a node must be in ascending order.

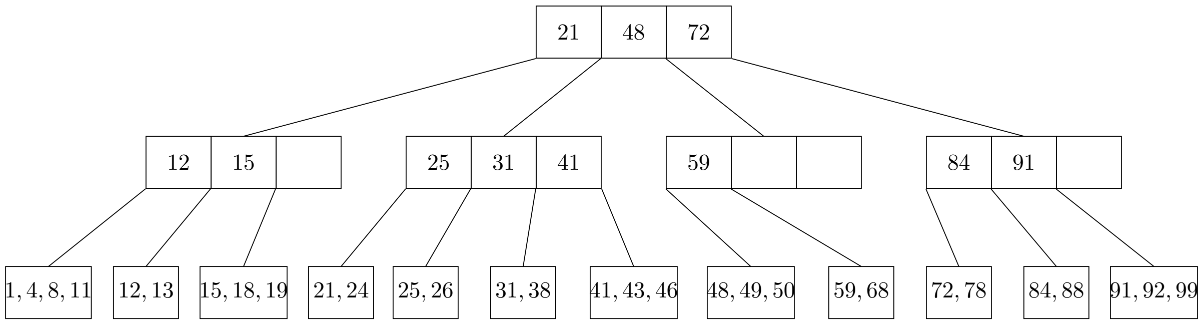


Figure 4.14: B-tree of order 4

A B-tree of order 4 is also known as a 2-3-4 tree; a B-tree of order 3 is also known as a 2-3 tree.

4.6.2 Operations

To insert a key into a node, we need to note the maximum number of values that the node can store. If the node isn't full, we can simply insert the key. However, there are some cases that need to be considered:

To insert 1, we find the location. However, since the leftmost node is full, we cannot insert it there. This can be solved by splitting the node into two nodes with two keys, then adjusting the information of the parent. Next, we try to insert 19. Since the rightmost node is full, we need to split it into two nodes with two children. Then, we continue splitting upwards to the root until we either reach the root node or find a node with fewer than two children.

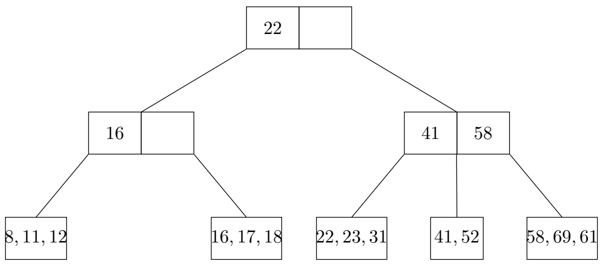


Figure 4.15: Original Tree

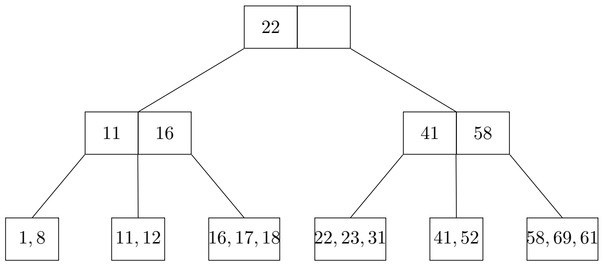


Figure 4.16: Insert 1

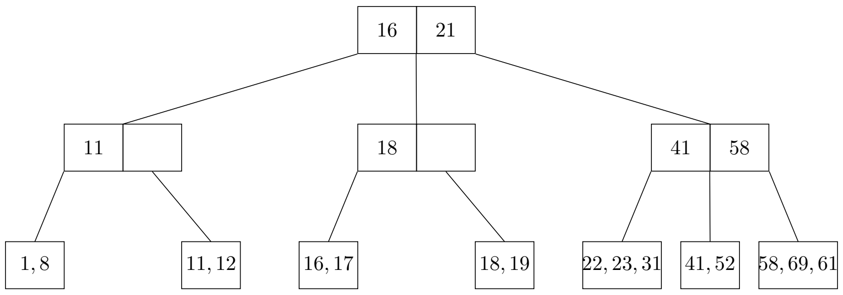


Figure 4.17: Insert 19

The depth of a B-tree is at most $\lceil \log_{\lceil \frac{m}{2} \rceil} n \rceil$. At each node along the path, we perform $O(\log m)$ work to determine which branch to take. An insertion or deletion could require $O(m)$ work to fix up all the information at the node. The worst case for insertion and deletion would be $O(m \log mn) = O\left(\frac{m}{\log m} \log n\right)$.

In summary, trees are used in operating systems, compiler design, and searching. In practice, all the balanced tree schemes are worse than the simple binary search tree, but this is acceptable.

Chapter 5

More on Tree

5.1 Tries

5.1.1 Definition

A trie, also called a digital tree, radix tree, or prefix tree, is a special type of tree used to store associative data structures. The name trie comes from its use for re**trie**val, because the trie can find a single word in a dictionary with only a prefix of the word.

For example, strings are stored in a top-to-bottom manner based on their prefixes in a trie. All prefixes of length 1 are stored at level 1, all prefixes of length 2 are stored at level 2, and so on. For the string set $S = \{\text{bear}, \text{bell}, \text{bid}, \text{bull}, \text{but}, \text{sell}, \text{stock}, \text{stop}\}$, we can have the tree structure as shown on the right.

This figure shows how the strings are stored in the trie. Inserting and deleting words in this tree is straightforward. For example, when typing "belt," it can be inserted in the subtree starting from `b -> e -> l`.

Unlike a binary search tree, no node in a trie stores the key associated with that node; instead, its position in the tree defines the key with which it is associated.

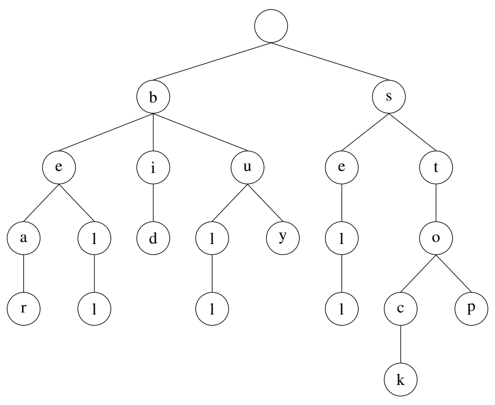


Figure 5.1: Tries

All the descendants of a node share a common prefix in the string associated with that node, and the root is associated with the empty string.

5.1.2 Applications

A trie can also be used to replace a hash table, offering the following advantages:

1. Faster lookup: Looking up data in a trie is faster than the worst case of a hash table. For a trie, the time complexity is $O(m)$, where m is the length of the search string. However, the time for an imperfect hash table would be $O(n)$, where n is the total number of strings.

2. No collisions: There are no collisions of different keys in a trie, unlike hash tables, where collisions can occur when two keys map to the same hash value.

3. No need for a hash function: In a trie, there is no need to provide a hash function or change it as more keys are added, unlike hash tables that require such adjustments.

4. Alphabetical ordering: A trie can provide an alphabetical ordering of the entries by key, making it useful for applications like autocomplete or lexicographical ordering.

A common application of a trie is storing a predictive text or autocomplete dictionary, such as those found on mobile phones. It is also used in web browsers, which autocomplete your text or show possible completions for the text you are typing. Additionally, a trie can act as an orthographic corrector, checking whether every word you type exists in a dictionary.

5.1.3 Analysis

A standard trie uses $O(n)$ space and supports searches, insertions, and deletions in time $O(dm)$, where n is the total size of the strings in S, m is the size of the string parameter of the operation, and d is the size of the alphabet.

5.2 B-Tree

Here we continue the discussion on B-Trees.

A B-Tree is a self-balanced search tree with multiple keys in each node and can have more than two children per node. The properties of B-Trees have been discussed in the previous chapter.

Following these properties, for example, a B-Tree of order 4 contains a maximum of 3 key values in a node and a maximum of 4 children for a node.

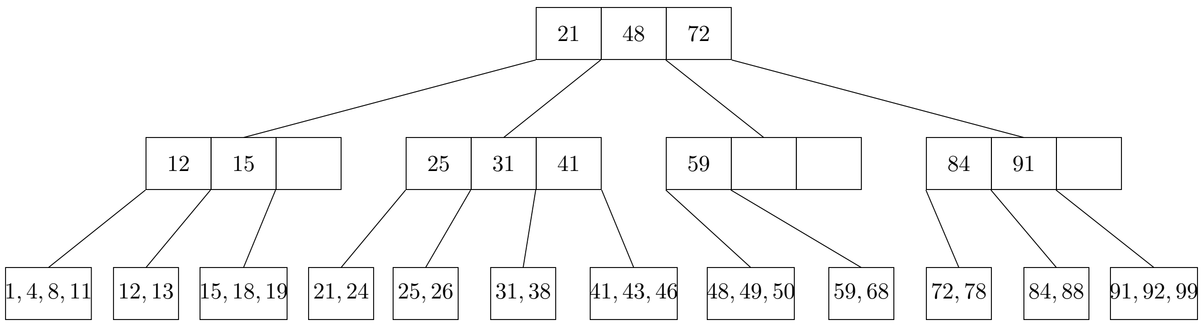


Figure 5.2: B-tree of order 4

5.2.1 Operations

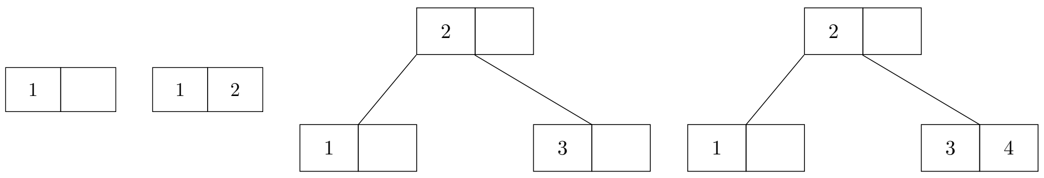
To search for a certain element in a B-Tree, we first read the value from the user. Then, we compare the search element with the first key value of the root node in the tree. If they match, we terminate the function immediately. If they do not match, we check whether the search element is smaller or larger, then move to the appropriate subtree where the element could be found. This process is done recursively until we either find the exact match or complete the comparisons with the last key value in a leaf node.

In a B-Tree, a new element must be added only at a leaf node. This means new key values are always attached to leaf nodes.

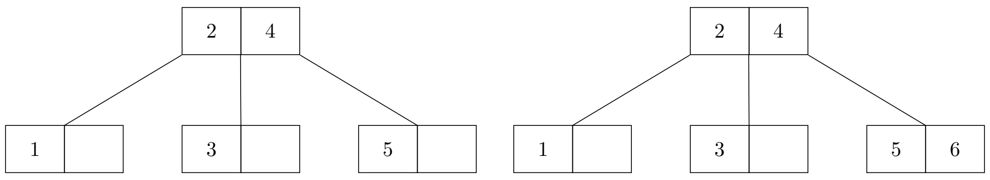
To perform an insertion, we first check if the tree is empty. If it is empty, we create a node with the new key value and insert it into the tree as the root node. Otherwise, we find the appropriate leaf node where the new key value can be added, using binary search tree logic. If the leaf node has an empty position, we add the new key value to the leaf node, maintaining the ascending order of key values within the node.

If the leaf node is already full, we split that leaf node by sending the middle value to its parent, and repeat this process until the value is placed into a node. If the split reaches the root, the middle value becomes the new root node, increasing the height of the tree by one.

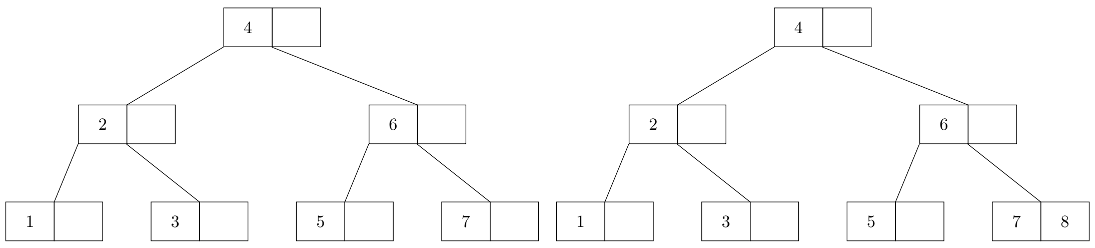
Following, we construct a B-Tree of order 3 by inserting numbers from 1 to 8.



For 1 and 2, we can directly do the insertion. For 3, since the node is full, we split the node by sending the middle value 2 to the parent node. Since there is an empty position, we can directly insert 4.



Since the rightmost node is full, we insert 5 by sending the middle value to the parent node and split the node. Then, we can directly insert 6.



Since the rightmost node is again full, we insert 7 by sending the middle value and splitting the node. Since the parent is also full, we further split it and send the middle value, in this case 4, to the parent node. To insert 8, it can be done directly.

To do deletion, if the key to be deleted is in a leaf and it contains more than the minimum number of keys, then this key can be deleted with no further action. If it is not a leaf, swap it with its successor under the natural order of the keys, then delete the key from the leaf.

If the node contains the minimum number of keys, consider the two immediate siblings of the node. If one of these siblings has more than the minimum number of keys, then redistribute one key from this sibling to the parent node, and one key from the parent to the deficient node. This is a rotation that balances the nodes.

If both immediate siblings have exactly the minimum number of keys, then merge the deficient node with one of the immediate sibling nodes and one key from the parent node. If this leaves the parent node with too few keys, then the process is propagated upward.

5.2.2 Analysis

The depth of a B-Tree is h, then

$$h \leq \log_t \frac{n+1}{2}, \quad t = \left\lceil \frac{m}{2} \right\rceil, \quad n = \text{the number of keys.}$$

The search, insertion, and deletion time is $O(t \log_t n)$.

Chapter 6

Hashing

From linear search, which takes $O(n)$ time, to binary search, which takes $O(\log n)$ time, the time is reduced. However, is there another data structure that allows better time? The answer is hashing.

6.1 Introduction

In some applications, fully utilizing the entire array is rare, leading to sparse arrays or sparse matrices. To avoid the multiplication operations required to calculate the index of an entry, we use an access table. This auxiliary table contains values like $0, n, 2n, 3n, \ldots, (m-1)n$. When referencing the rectangular array, the index for entry $[i, j]$ is computed by accessing the value at position i in the auxiliary table, adding j, and using the resulting position.

A table with index set I and base type T can be viewed as a function from I to T, along with two main operations:

- Table access: Evaluate the function at any index in I.

- Table assignment: Modify the function by changing the value at a specified index in I to a new value.

Insertion involves adding a new element x to the index set I and defining the corresponding value for the function at x. Deletion removes an element x from the index set I, restricting the function to the resulting smaller domain.

However, array indices are not natural identifiers for the items we want to store, access, and retrieve. This limitation is overcome by using hashing.

6.1.1 Hashing

Hashing is a technique used to perform insertions, deletions, and lookups in constant average time. However, hashing does not efficiently support tree operations that require ordering information among elements. There are three important components of hashing:

1. Hash function: to generate a key;

2. Hash table: to store the elements;

3. Collision resolution: to resolve conflicts when two keys hash to the same index.

A hash table is an abstract data type that allows storing and retrieving elements in constant time, $O(1)$. This is true when the indices are known and the value at the target index of a store operation can be discarded. Without a complete set of items, we cannot determine the index of an element in a sorted list.

To solve this problem, we use the item as a key and convert it into unique integers that can be used as array indices. Since the index integer is not known at the entry of an item, it would be helpful if the item itself could be used as a key to index the cell where it will be stored. For example, to store names in an array, rather than assigning arbitrary indices, we can sum the ASCII values of each character in the key (e.g., $\mathrm{a} = 1, \mathrm{b} = 2, \ldots$).

6.1.2 General Idea

As defined previously, a function that performs the conversion is called a hash function. The conversion process is called hashing, and the storage structure used is called a hash table or scatter-storage.

In general, a hash table is an array of fixed size that contains keys. Each key is associated with a value and mapped to a number in the range 0 to H_SIZE − 1, placing it in the corresponding cell.

The mapping of keys to indices is done by the hash function, which ideally should be simple to compute and should ensure that distinct keys are placed in different cells. However, this is challenging because there are a finite number of cells and a virtually limitless number of potential keys. Therefore, the goal is to design a hash function that distributes the keys as evenly (uniformly) as possible among the cells.

There are several important issues that need to be addressed, which will be discussed in this chapter:

1. Choosing the hash function.
2. Handling collisions.
3. Handling deletions.

6.2 Hash Table

The idea of a hash table is to allow many of the different possible keys that might occur to be mapped to the same location in an array under the action of the index function. This is sometimes referred to as scatter-storage or key-transformation.

A hash function takes a key and maps it to an index in the array. However, two or more records may be mapped to the same location, leading to a collision. Therefore, a collision resolution procedure must be devised to handle this situation.

6.3 Hash Function

6.3.1 Analysis

The two principal criteria in selecting a hash function are that:

1. it should be easy and quick to compute,
2. it should achieve an even distribution of the keys that actually occur across the range of indices.

If the input keys are integers, then simply returning `key mod H_SIZE` is generally a reasonable strategy. For example, `student_ID mod 10000` would be a reasonable strategy. Also, it is usually a good idea to ensure that the table size is prime. When the input keys are random integers, this function is simple to compute and also distributes the keys evenly.

6.3.2 Truncation

Truncation can be done by ignoring part of the key and using the remaining part directly as the index (considering non-numeric fields as their numerical codes). For example, if the keys are eight-digit integers and the hash table has 1000 locations, then the first, second, and fifth digits from the right make the hash function, so that 62538194 maps to 394. Although this method is fast, it often fails to distribute the keys evenly through the table.

6.3.3 Folding

We can also partition the key into several parts and combine the parts in a convenient way (often using addition or multiplication) to obtain the index. For example, we can map 62538194 to $625 + 381 + 94 = 1100$, which is truncated to 100.

6.3.4 Modular Arithmetic

We can use modular arithmetic to convert the key into the index that we want. We can simply use the ASCII values of all the characters in the string, and then return the `result mod H_SIZE`.

For example, 'abcd' mod 100 $= (64 + 65 + 66 + 67)\%100 = 62$.

We can also take only 3 characters and use `hash_val = key[0] + 27*key[1] + 729*key[2]`, and then again return the `hash_val mod H_SIZE`, assuming that the key has at least two characters plus the NULL terminator. If the three characters are random, and the table size is 10007, then we would have a reasonably equitable distribution. However, since English is not random, the percentage of the table being hashed to could be less than expected.

We can improve the hash function by using `hash_val = hash_val << 5 + *key++`, then returning `hash_val mod H_SIZE`. This hash function involves all the characters in the key by computing:

$$\sum_{i=0}^{\text{keySize - 1}} \text{key[keySize - i]} \times 32^i,$$

which follows Horner's Rule.

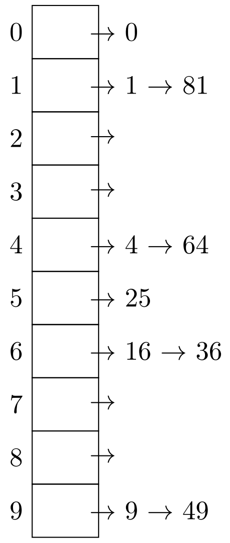
It is common to not use all the characters in the key, since the length and properties of the key would influence the choice.

6.4 Collision Resolution

Though a hash table is an efficient data structure, there could be cases where different elements share the same index, causing a collision. Thus, we need to handle such collisions, and there are several resolutions.

6.4.1 Open Hashing

The first strategy, commonly known as open hashing or separate chaining, is to keep a list of all elements that hash to the same value.



To illustrate, we assume that the keys are the first 10 perfect squares, where the hashing function is given by hash$(x) = x \mod 10$.

In open hashing, each node contains a linked list, allowing multiple elements to be stored at the same index.

We can perform **find** in open hashing by using the hash function to determine which list to traverse. Then, we traverse this list in the normal manner, returning the position where the item is found.

To perform **insertion**, we first traverse the list to check whether the element is already present. If it is a new element, it is inserted either at the front or the end of the list. Often, new elements are inserted at the front, as this is convenient and because recently inserted elements are frequently the most likely to be accessed again soon.

The **deletion** routine is straightforward, similar to linked lists. After performing a find operation, we delete the item as we would in a linked list.

The use of linked storage can save a considerable amount of space. It allows for simple and efficient collision handling, and the size of the hash table no longer needs to exceed the number of records we have. Additionally, deletion is faster compared to other methods.

However, all the pointers (links) require space, and even if the number of records is small, the space used is comparatively larger. Thus, it takes longer time to allocate the new cells, and it requires the implementation of another data structure. Thus, we introduce Closed Hashing.

6.4.2 Closed Hashing

In closed hashing, if a collision occurs, alternate cells are tried until an empty cell is found.

For example, cells $h_0(x), h_1(x), \cdots$ are tried in succession where $h_i(x) = (\text{hash} + f(i)) \mod \text{H\_SIZE}$, with $f(0) = 0$.

The function $f(i)$ is called the collision resolution strategy. Because all the data goes inside the table, a bigger table is needed for closed hashing than for open hashing. Generally, the load factor should be below $l = 0.5$ for closed hashing.

Again, we want to perform insertion. An array must be declared that will hold the hash table. Then, all locations in the array need to be initialized to show that they are empty. To insert a record, the hash function for the key is first calculated. If the corresponding location is empty, then the record can be inserted. Otherwise, insertion of the new record would not be allowed, which is what we call a collision.

For the find operation, we again calculate the key first. If the desired record is in the corresponding location, then the retrieval has succeeded. Otherwise, we follow the same procedure as for collision resolution, examining all locations until we find the correct record. However, if the position is empty, then no record with the given key is in the table, and the search is thus unsuccessful.

What we haven't discussed yet is the way to resolve collisions. In general, there are three methods.

Linear Probing

This is the simplest method to resolve a collision. Starting with the hash address where the collision happens, do a sequential search for the desired key or an empty location.

However, data could become clustered in this method. That is, records start to appear in long strings of adjacent positions with gaps between the strings.

For example, given $x = 89, 18, 49, 58, 69$, with $\text{hash}(x) = x \mod 10$, we can perform insertion as shown on the right.

For values 89 and 18, they can be inserted directly since the corresponding cell for the address is empty. However, when inserting 49, the key value 9 is repeated. Then it finds the next available cell, which in this case would be at position 0. The same applies to 58 and 69.

	Empty Table	89	18	49	58	69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Although they can all be inserted, the time to search for an empty cell may be long. Since the records are not distributed uniformly and become progressively more unbalanced, this leads to the problem of primary clustering. This problem is essentially one of instability. If a few keys happen randomly to be near each other, then it becomes more and more likely that other keys will join the cluster.

Assuming a very large table and that each probe is independent of the previous probes, the number of probes for a successful search is equal to the number of probes required when the particular element is inserted. When an element is inserted, it is done as a result of an unsuccessful search. We can use the cost of an unsuccessful search to compute the average cost of a successful search.

Since the fraction of empty cells is $1 - \lambda$, the number of cells we expect to probe is $\frac{1}{1-\lambda}$.

Then, it can be shown that the expected number of probes using linear probing is roughly

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

for insertions and unsuccessful searches. It takes roughly

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

for successful searches. Here, λ is the ratio of the number of elements in the hash table to the table size.

This is inefficient since, intuitively, we need to keep probing, which takes longer. As λ increases, the expected number of probes also increases. This issue becomes more significant if the table is expected to be more than half-full.

Thus, we have quadratic probing.

Quadratic Probing

Quadratic probing avoids the primary clustering problem of linear probing. If there is a collision at hash address H, the method called quadratic probing looks in the table at locations $h+0, h+1, h+4, h+9, \cdots$, that is, at location $h+i^2$ for $i = 0, 1, 2, \cdots$.

This reduces clustering, but it is obvious that it will not probe all locations in the table. The idea is that if quadratic probing is used and the table size is prime, then a new element can always be inserted if the table is at least half empty.

Here we use the same example. Now we use quadratic probing instead. After 18 and 89 being inserted, for 49, it goes to $(9+1^2) \mod 10 = 0$ since location 9 is occupied, for 58, it goes to $(8+2^2) \mod 10 = 2$, and for 69 it goes to $(9+2^2) \mod 10 = 3$.
Note that if the table is even more than half full, insertion could fail. It is also crucial that the table size is a prime number. Otherwise, the number of alternate locations can be severely reduced.

	Empty Table	89	18	49	58	69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Standard deletion cannot be performed in a closed hash table since the cell might have caused a collision to pass it. Simply put, deleting an entry could break the probing chain, causing searches to miss other items. Thus, lazy deletion is required.

In lazy deletion, we delete an entry by placing a special marker or key in the deleted position. This marker indicates that the position is free for future insertions but should not be treated as empty when searching for other items — ensuring the probing sequence remains unbroken.

Random Probing

Rather than having the increment depend on the number of probes already made, we can let it be a simple function of the key itself. For example, we could truncate the key to a single character and use its code as the increment.

For random probing, we use a pseudo-random number generator to obtain the increment. The generator should always produce the same sequence when given the same seed. This method is excellent at avoiding clustering, but it is likely to be slower than others.

Double Hashing

Another closed hashing method is double hashing. For double hashing, one popular choice is:

$$f(i) = i \times h_2(x).$$

We apply a second hash function to x and probe at distances $h_2(x), 2h_2(x), \ldots,$ and so on. However, a poor choice of $h_2(x)$ could be disastrous. The function must never evaluate to zero, and it needs to ensure that all cells can be probed.

For instance, the obvious choice $h_2(x) = x \mod 9$ would not help if 99 were inserted into the input in the previous example. A function such as:

$$h_2(x) = R - (x \mod R),$$

with R being a prime number smaller than H_SIZE, works well. One may also perform triple hashing, and so on.

Again, we perform insertion for $x = 89, 18, 49, 58, 69$ using double hashing. Here, we have $h_2(x) = R - (x \mod R)$, with $R = 7$.
After inserting 89 and 18, 49 causes a collision. Then we use $h_2(49) = 7 - (49 \mod 7) = 7$, so it goes to $(7 + 49) \mod 10 = 6$. Inserting 58 also causes a collision, then we have $h_2(58) = 7 - (58 \mod 7) = 5$, so it goes to $(5 + 58) \mod 10 = 3$.

	Empty Table	89	18	49	58	69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

6.4.3 Rehashing

There could be cases where the hash table becomes too full, causing the running time for operations to deteriorate, especially when there are too many deletions intermixed with insertions.
To solve this issue, we can build another table that is about twice as large with an associated new hash function. Then, we scan the entire original hash table, compute the new hash value for each element, and insert it into the new table. This process is known as rehashing.
For example, for a table of size 7, let $h(x) = x \mod 7$. After inserting $x = 13, 15, 24, 6$, we insert 23, which makes the table now 70% full. Thus, we create a new table with size 17. The new hash function is then $h(x) = x \mod 17$. The old table is scanned, and elements 6, 23, 24, 13, and 15 are then inserted sequentially.
The running time is $O(n)$, which is expensive. Therefore, it should not be done too frequently. This leads us to the concept of extendible hashing.

0	6
1	15
2	
3	24
4	
5	
6	13

0	6
1	15
2	23
3	24
4	
5	
6	13

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	
17	

When the amount of data is too large to fit in main memory and must be stored on disk, we minimize disk access by using a tree.

In definition, the root of the tree is called the *directory*, and the leaves are called *buckets*. The number of bits used by the root is called the *global depth*, and the *bucket size* is the maximum number of elements that a leaf can contain.

Suppose that our data consists of several 6-bit integers. With global depth D equal to 2, the root of the tree contains 4 pointers determined by the leading two bits of the data. Each leaf can hold up to $m = 4$ elements.

For example, we have the following tree. To insert 36, we first convert it to binary 100100_2. The hash function here is the D most significant bits (MSBs), which is the 2 MSBs in this case. Since the bucket of 00 is full, and the global depth is equal to the bucket depth, the directory is split, and the roots now have $D = 3$.

Then, we update the content by inserting 100100_2 into the directory 100. Note that for the same bucket, there could be more than one pointer pointing to it.

Next, to insert 000000_2, we again encounter a collision. However, since the global depth is now larger than the bucket depth, the bucket itself is split instead of the directory. The element can then be inserted.

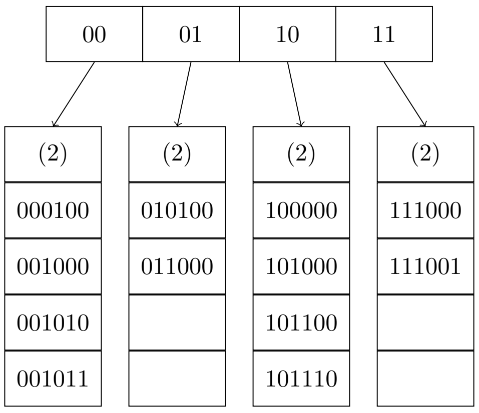


Figure 6.1: Original

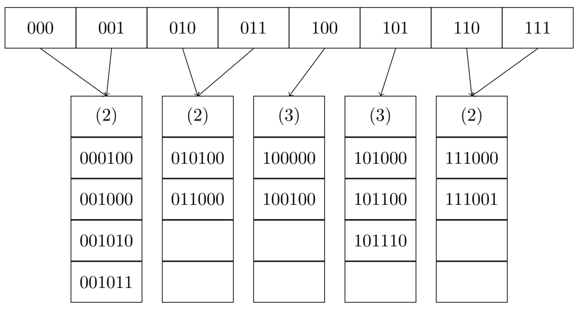


Figure 6.2: Insertion 100100

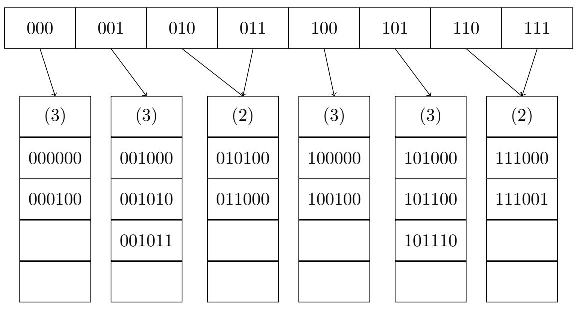


Figure 6.3: Insertion 000000

It is possible that several directory splits will be required if the elements in a leaf agree in more than $D + 1$ leading bits, i.e., overflow. For example, to insert $111010_2, 111011_2, 111100_2$, the directory size must be increased to 4.

This algorithm does not work when there are more than m duplicates. It is important for the bits to be fairly random.

In summary, hash tables can be used to implement the insertion and find operations in constant average time. It is especially important to pay attention to details such as the load factor when using hash tables. The choice of hash function is also crucial, especially when the key is not a short string or integer.

For open hashing, the load factor should be 1, meaning all the cells can be filled. For closed hashing, the load factor should not exceed 0.5, unless unavoidable. It is not possible to find the minimum or maximum since there is no sorting order.

In applications, hash tables can be used in compilers to keep track of declared variables in source code. They are useful for any graph theory problem where nodes have real names instead of numbers. Additionally, hash tables are commonly used in programs that play games or even online spelling checkers.

Chapter 7

Heaps

Here, we continue the discussion on queues.

7.1 Introduction

In some applications, a simple queue may not be the best strategy for completing jobs. Problems arise when small jobs take longer to finish, or important jobs are not processed first.

That's why we have heaps, which implement a priority queue. Unlike a regular queue, which follows the first-in, first-out (FIFO) principle, a priority queue selects an entry based on specific properties and places it at the front to be processed first.

For example, in a job queue, various algorithms can be implemented to manage tasks, such as first-come, first-served, shortest-job-first, longest-job-first, priority-based scheduling, or even a combination of these methods.

7.1.1 Priority Queue

A priority queue consists of entries, each containing a key called the priority of the entry. A priority queue supports two primary operations in addition to the usual creation, size, full, and empty checks: **Insert** and **Delete_Min**. Insertion is straightforward, while Delete_Min finds, returns, and removes the entry with the highest priority. If all entries have equal priorities, the queue follows the FIFO (first-in, first-out) rule.

Several possible implementations include a simple linked list, a sorted contiguous list, an unsorted list, and a binary search tree.

7.2 Binary Heaps

Binary Heaps (or Heaps) have two properties: the **structure property** and the **heap order property**. For the **structure property**, the heap must be a **complete binary tree**.

As with AVL trees, an operation on a heap can destroy one of these properties. Therefore, a heap operation must not terminate until all heap properties are restored.

7.2.1 Structure Property

A heap is a binary tree that is completely filled (a complete tree), with the possible exception of the bottom level, which is filled from left to right.

A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes. This implies that the height of a complete binary tree is $\lfloor \log n \rfloor$, which is clearly $O(\log n)$.

Because a complete binary tree is so regular, it can be represented in an array, and no pointers are necessary. For example, to represent the tree on the left, we can have:

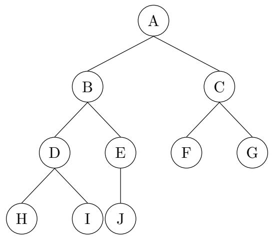


Figure 7.1: Binary Heap

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Table 7.1: Implementation

For any element in array position i, the left child is in position $2i$, the right child is in the cell after the left child $(2i + 1)$, and the parent is in position $\lfloor \frac{i}{2} \rfloor$. Thus, not only are pointers not required, but the operations required to traverse the tree are extremely simple.

For example, A is in position 1, thus it has no parent. Its children will be in positions 2 and 3, which are B and C. For B in position 2, its parent will be 1, which is A. Its children will be in positions 4 and 5, which are D and E.

The problem is that the estimation of the maximum heap size is required in advance.

7.2.2 Heap Order Property

This property allows operations to be performed quickly. For a heap, the smallest element should be at the root so that the operation to remove it is efficient. By the heap order property, the minimum element can always be found at the root. Thus, we have the extra operation, find_min, which can be performed in constant time $O(1)$.

Since we want to be able to find the minimum quickly, it makes sense that the smallest element should be at the root. If we consider that any subtree should also be a heap, then any node must be smaller than all of its descendants. By applying this logic, we arrive at the heap order property:

In a heap, for every node X, the key in the parent of X is smaller than or equal to the key in X, except for the root.

7.3 Operations

To perform insertions, we create a hole in the next available location. If x can be placed in the hole without violating the heap order, then we do so, and the insertion is complete. Otherwise, we slide the element from the parent node of the hole into the hole, thus bubbling the hole up toward the root. We continue this process until x can be placed in the hole. This strategy is called **percolate up**.

For example, we use the following method to insert 14 into a heap.

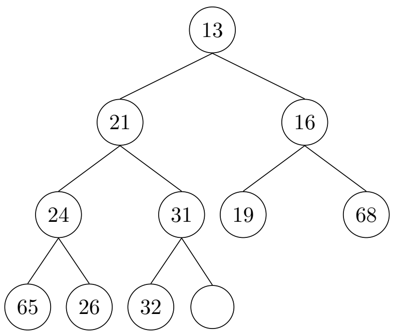


Figure 7.2: Create hole

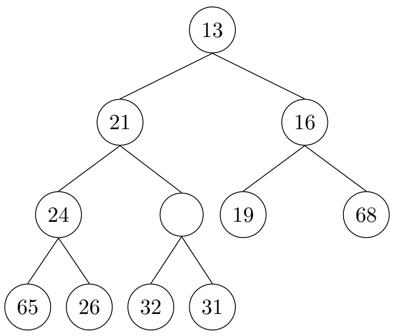


Figure 7.3: Compare

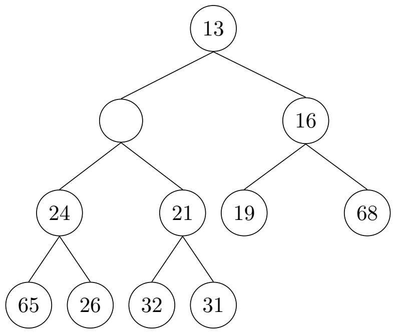


Figure 7.4: Compare

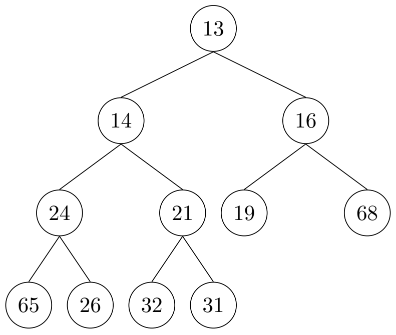
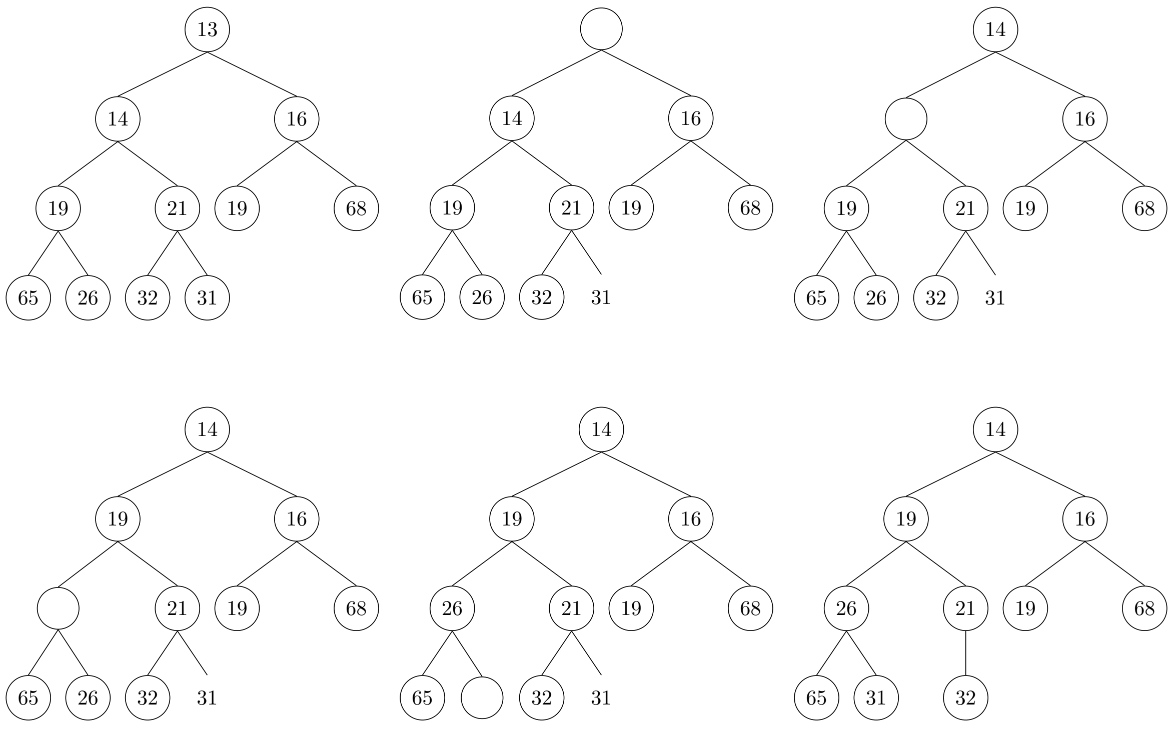


Figure 7.5: Insert 14

The time to perform the insertion can be as much as $O(\log n)$ if the element to be inserted is the new minimum and is percolated all the way to the root. It has been shown before that 2.607 comparisons are required on average to perform an insertion. On average, the insertion moves an element up 1.607 levels.

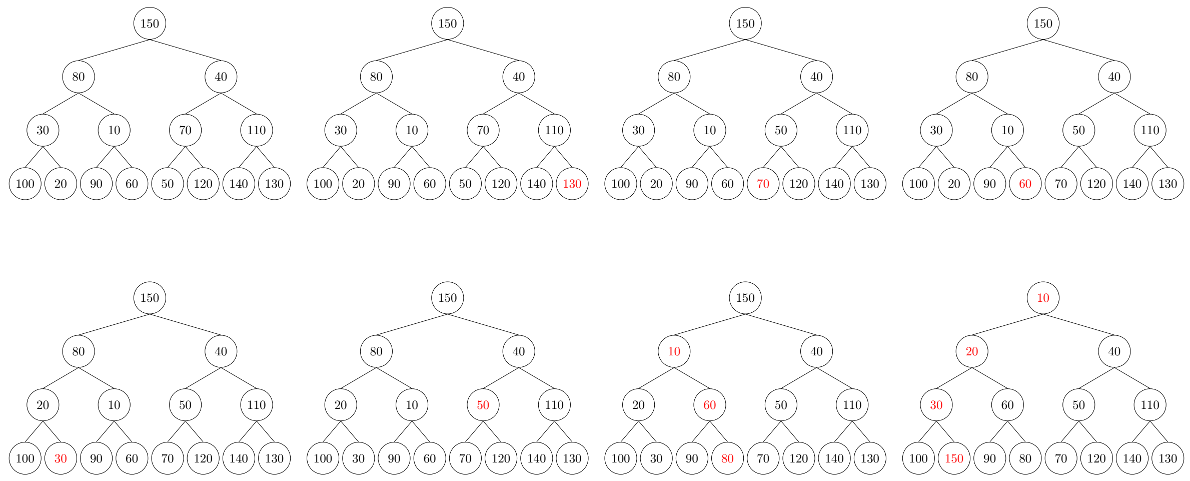
Deletions are handled in a manner similar to insertions. We only delete from the top since it is the smallest value. When the root is deleted, a hole is created. Then, we slide the smaller of the hole's children into the hole, thus pushing the hole down one level. We repeat this step until x can be placed in the hole. Thus, our action is to place x in its correct spot along a path from the root containing the minimum children. The rearranging will typically take less than $O(\log n)$.

For example, we use the following method to perform deletion:



There are some other heap operations. We can find the minimum in constant time. However, it would be difficult, though possible, to find the maximum, given that there is no ordering information.

To build a heap, we take n keys and place them into an empty heap. We could then perform n successive inserts. This will take $O(n)$ on average but $O(n \log n)$ in the worst case. Another way is to place the n keys into the tree in any order, then perform percolate-down on half of the nodes. For example, to insert 15 keys, we can perform percolate-down starting from the 7th node.



In this process, we start with the last node that is not a leaf node, compare it with its left and right children, and interchange the node with the larger of its two children. We continue this process with the node until it becomes a leaf node or until the heap property is restored.

To perform a k-selection, we could find the k-th smallest or largest element in a set. To build a heap, the average time complexity is $O(n)$, and the worst-case time complexity is $O(n \log n)$. To delete from a heap, the time complexity is $O(\log n)$. Hence, the total runtime for the k-selection problem is $O(m + k \log n)$.

For small k, the running time is dominated by the heap-building operation, making it $O(n)$. For larger values of k, the running time is $O(k \log n)$.

Alternatively, we could build a smaller heap tree of k elements and then compare the remaining entries against the heap. If the new element is larger, it replaces the root; otherwise, it is discarded. To build a heap of k elements, the time complexity is $O(k)$. The time to process each of the remaining elements is $O(1)$. To test if an element should be added to the heap, we incur an additional $O(\log k)$ to delete the root and insert the new element if necessary.

Thus, the total time complexity is $O(k + (n - k) \log k) = O(n \log k)$. This algorithm also provides a bound of $n \log n$ for finding the median.

Chapter 8

Sorting

Sorting is simply the process of ordering data in a consistent manner, such as organizing cards, telephone name lists, student name lists, etc.

Each element is usually part of a collection of data called a record. Each record contains a key, which is the value to be sorted, while the remainder of the record consists of satellite data. Here, we have two assumptions: the keys are integers, and the sorting process uses internal memory.

There are several algorithms to sort in $O(n^2)$, such as insertion sort. There is also an algorithm like Shell sort, which is very simple to code, runs in $O(n^2)$, and is efficient in practice. Additionally, there are some slightly more complicated sorting algorithms that take $O(n \log n)$. However, any general-purpose sorting algorithm requires $\Omega(n \log n)$ comparisons.

8.1 Introduction

8.1.1 Preliminaries

In internal sort, an array containing the elements and an integer containing the number of elements will be passed to the algorithm.

We assume that N, the number of elements passed to our sorting routines, has already been checked and is legal.

We require the existence of the $<$ and $>$ operators, which can be used to place a consistent ordering on the input.

8.1.2 Operations

In sorting, there are several operations.

Permutation

A permutation of a finite set S is an ordered sequence of all the elements of S, with each element appearing exactly once. A k-permutation of S is an ordered sequence of k elements of S, with no element appearing more than once in the sequence.

For example, for $S = \{a, b, c\}$, there are 6 permutations. For a set of n elements, there are $n!$ permutations.

Inversion

An inversion in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$.

For example, the input list $34, 8, 64, 51, 32, 21$ has nine inversions, namely: (34,8), (34,32), (34,21), (64,51), (64,32), (64,21), (51,32), (51,21) and (32,21). Notice that this is exactly the number of swaps that need to be performed by insertion sort.

8.2 Bubble Sort

8.2.1 Operation

Bubble sort is done by scanning the list from one end to the other, and whenever a pair of adjacent keys is found to be out of order, they are swapped. In this pass, the larger key in the list will have bubbled to the end, but earlier keys may still be out of order.

Bubble sort is probably the easiest algorithm to implement but the most time-consuming of all the algorithms, other than pure random permutation. The basic idea underlying bubble sort is to pass through the file sequentially several times.

In bubble sort, each pass consists of comparing each element in the file with its successor and interchanging the two elements if they are not in proper order. After each pass, the largest element $x[n-i]$ is in its proper position within the sorting array.

Original	34	8	64	51	32	21	Number of Exchanges
After $p = 1$	8	34	21	64	51	32	4
After $p = 2$	8	21	34	32	64	51	3
After $p = 3$	8	21	32	34	51	64	2
After $p = 4$	8	21	32	34	51	64	0
After $p = 5$	8	21	32	34	51	64	0
After $p = 6$	8	21	32	34	51	64	0

In this example, for the first pass, it begins from the right. Since $21 > 32$, it swaps, then we have $34, 8, 64, 51, 21, 32$. Then we check with 51 and 21, they are out of order, so we swap again, then we have $34, 8, 64, 21, 51, 32$. Recursively doing so, we get the sequence $8, 34, 21, 64, 51, 32$. The sorting will stop when $p = 6$, which is the number of elements to be sorted.

8.2.2 Analysis

There are in total $n-1$ passes and $n-1$ comparisons on each pass without the improvement. Thus, the total number of comparisons is $(n-1)^2 = n^2 - 2n + 1$, which is $O(n^2)$.

We can improve this algorithm by stopping the swap when the number of exchanges is equal to 0, i.e., all the elements are in proper order. With the improvement, the sorting will be $(n-1)+(n-2)+\cdots+1 = \frac{n(n+1)}{2}$, which is also $O(n^2)$.

The number of interchanges depends on the original order of the file. However, the number of interchanges cannot be greater than the number of comparisons. It is likely that the number of interchanges, rather than the number of comparisons, takes up the most time in the program's execution.

For bubble sort, it requires little additional space. We only need one additional record to hold the temporary value for interchanging and several simple integer variables. It is $O(n)$ in the case that the file is completely sorted (or almost completely sorted), since only one pass of $n-1$ comparisons (and no interchanges) is necessary to establish that the file is sorted.

8.3 Insertion Sort

8.3.1 Operation

Insertion sort again consists of $n-1$ passes. For pass $p=2$ through n, insertion sort ensures that the elements in positions 1 through p are in sorted order. Insertion sort makes use of the fact that elements in positions 1 through $p-1$ are already known to be in sorted order.

Initially, $x[0]$ may be thought of as a sorted file of one element. After each repetition of the loop, the elements $x[0]$ through $x[k]$ are in order.

Original	34	8	64	51	32	21	Positions Moved
After $p=1$	34	8	64	51	32	21	0
After $p=2$	8	34	64	51	32	21	1
After $p=3$	8	34	64	51	32	21	0
After $p=4$	8	34	51	64	32	21	1
After $p=5$	8	32	34	51	64	21	3
After $p=6$	8	21	32	34	51	64	4

In this example, for the first pass, we treat 34 as sorted, thus no move is needed. Then we 'insert' 8, since it is smaller than 34, it is moved before 34. By doing so recursively, we can have a sorted list.

8.3.2 Analysis

The simple insertion sort may be viewed as a general selection sort in which the priority queue is implemented as an ordered array. Only the preprocessing phase of inserting the elements into the priority queue is necessary. Once the elements have been inserted, they are already sorted, so no selection is necessary.

If the initial file is sorted, only one comparison is made on each pass, so the sort is $O(n)$. However, if the file is initially sorted in reverse order, the sort is $O(n^2)$, since the total number of comparisons is $(n-1)+(n-2)+\cdots+2+1=\frac{(n-1)n}{2}$, which is still $O(n^2)$.

The simple insertion sort is still usually better than the bubble sort. The closer the file is to sorted order, the more efficient the simple insertion sort becomes. The average number of comparisons in the simple insertion sort is also $O(n^2)$, considering all possible permutations of the input array.

The space requirements for the sort consist of only one temporary variable y. Insertion sort makes $O(n^2)$ comparisons of keys and $O(n^2)$ movements of entries. It makes $\frac{n^2}{4}+O(n)$ comparisons of keys and movements of entries when sorting a list of length n in random order.

To improve, we can use a binary search. This reduces the total number of comparisons from $O(n^2)$ to $O(n\log n)$. However, the moving operation still requires $O(n^2)$ time. So the binary search does not significantly improve the overall time requirement.

We can also improve the algorithm by using list insertion. This reduces the time required for insertion but not the time required for searching for the proper position. The average number of inversions in an array of n distinct numbers is $\frac{n(n-1)}{4}$.

For any list L of numbers, consider L_r, which is the list in reversed order. Consider any pair of two numbers in the list (x,y) with $y>x$. In exactly one of L and L_r, this ordered pair represents an inversion. The total number of these pairs in a list L and its reverse L_r is $\frac{n(n-1)}{2}$. On average, it is half of the above.

Any algorithm that sorts by exchanging adjacent elements requires $\Omega(n^2)$ time on average. The average number of inversions is initially $\Omega(n^2)$, and since each swap removes only one inversion, $\Omega(n^2)$ swaps are required.

8.4 Selection Sort

8.4.1 Operation

Selection sort is also called Straight Selection or Push-Down Sort. A selection sort is one in which successive elements are selected in order and placed into their proper sorted positions. The elements of the input may have to be preprocessed to make the ordered selection possible.

At the first stage, one scans the list to find the entry that comes last in the order. This entry is then interchanged with the entry in the last position. By omitting the last entry, we can repeat the process on the shorter list.

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	34	8	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	4
After $p = 3$	8	21	64	51	32	34	2
After $p = 4$	8	21	32	51	64	34	2
After $p = 5$	8	21	32	34	64	51	1
After $p = 6$	8	21	32	34	51	64	0

Here, we scan from the first element that comes in the order, which will give us the same sorted list. In the first scan, 8 is found to be the smallest element, so we swap it with the element in the first position. Then, 21 is found to be the second smallest element, so it is swapped with the element in the second position. By doing so recursively, we obtain a sorted list.

8.4.2 Analysis

Here we can do a simple comparison:

	Selection	Insertion (average)
Assignments of entries	$3n + O(1)$	$0.25n^2 + O(n)$
Comparisons of keys	$0.5n^2 + O(n)$	$0.25n^2 + O(n)$

The algorithm consists entirely of a selection phase in which the largest of the remaining elements L is repeatedly placed in its proper position i at the end of the array. To do so, L is interchanged with the element $x[i]$. The initial n-element priority queue is reduced by one element after each selection.

The first pass makes $n - 1$ comparisons, the second pass makes $n - 2$, and so on. Therefore, the total number of comparisons is $\frac{n(n-1)}{2} = O(n^2)$.

The number of interchanges is always $n - 1$ unless a test is added to prevent the interchanging of an element with itself.

There is little additional storage required except to hold a few temporary variables. The sort may be categorized as $O(n^2)$, although it is faster than bubble sort. There is no improvement if the input file is completely sorted or unsorted, since the testing proceeds to completion without regard to the makeup of the file.

8.5 Shell Sort

8.5.1 Operation

Shell sort is also called Diminishing Increment Sort. Selection sort moves the entries very efficiently, but there are many redundant comparisons. Also, even in the best case, insertion sort does the minimum number of comparisons, but it is inefficient in moving entries only one place at a time.

If we modify the comparison method so that it first compares keys far apart, then it could sort the entries that are far apart. Afterward, the entries closer together would be sorted, and finally, the increment between keys being compared would be reduced to 1, ensuring that the list is completely in order. This is what we call shell sort.

One requirement that is intuitively clear is that the elements of the increment sequence should be relatively prime. This guarantees that successive iterations intermingle subfiles so that the entire file is indeed almost sorted when the increment equals 1 in the last iteration.

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 2-sort	11	12	15	17	28	35	58	41	81	75	94	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

At first, we perform sorting with an increment of 4, so elements in positions 1, 6, and 11 are sorted; 2, 7, and 12 are sorted; and 3, 8, and 13 are sorted. By doing so recursively, we obtain a sorted list.

Since the first increment used by Shell sort is large, the individual subfiles are quite small, making the simple insertion sorts on those subfiles fairly fast. Each sort of a subfile causes the entire file to become more nearly sorted. Although successive passes of Shell sort use smaller increments and therefore deal with larger subfiles, those subfiles are almost sorted due to the actions of previous passes. Thus, the insertion sorts on those subfiles are also quite efficient.

8.5.2 Analysis

If a file is partially sorted using an increment k and is subsequently partially sorted using an increment j, the file remains partially sorted on the increment k. Hence, subsequent partial sorts do not disturb the earlier ones.

The analysis of Shell sort is difficult at best. Empirical studies on Shell sort show that when n is large, the running time is in the range of $n^{1.25}$ to $1.6n^{1.25}$. It has been shown that the order of Shell sort can be approximated by $O(n(\log n)^2)$ if an appropriate sequence of increments is used. For other series of increments, the running time can be proven to be $O(n^{1.25})$.

Empirical data indicate that the running time is of the form $a \times n^b$, where a is between 1.1 and 1.7, and b is approximately 1.26, or of the form $c \times n(\log n)^2 - d \times n \times \log n$, where c is approximately 0.3 and d is between 1.2 and 1.75.

In general, Shell sort is recommended for moderately sized files of several hundred elements.

Knuth recommends choosing increments as follows:

- Define a function h recursively so that $h(1) = 1$ and $h(i + 1) = 3 \times h(i) + 1$.

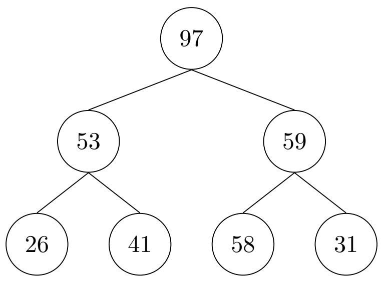
- Let x be the smallest integer such that $h(x) \geq n$, and set `numinc`, the number of increments, to $x - 2$ and `incrmnts[i]` to $h(\texttt{numinc} - i + 1)$ for i from 1 to `numinc`.

8.6 Heap Sort

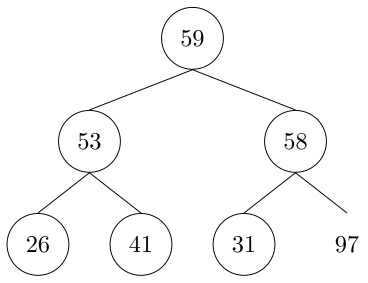
Heap sort is similar to an AVL tree. Priority queues can be used to sort in $O(n \log n)$ time, which gives the best big-O running time we have seen so far.

To perform heap sort, we build a max binary heap of n elements. This stage takes $O(n)$ time. We then perform n `delete_max` operations. The elements leave the heap, smallest first, in sorted order. By recording these elements in a second array and then copying the array back, we sort the n elements. Since each `delete_max` operation takes $O(\log n)$ time, the total running time is $O(n \log n)$.

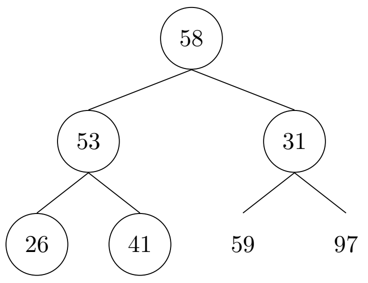
The main problem with this algorithm is that it uses an extra array. Thus, the memory requirement is doubled. To address this, we use the last cell for storing the array. Since after each `delete_max`, the heap shrinks by 1, the cell that was last in the heap can be used to store the element that was just deleted.



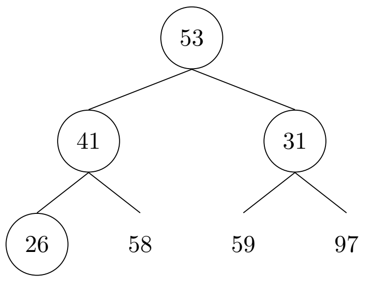
97, 53, 59, 26, 41, 58, 31



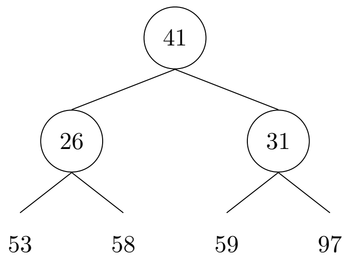
53, 59, 26, 41, 58, 31, 97



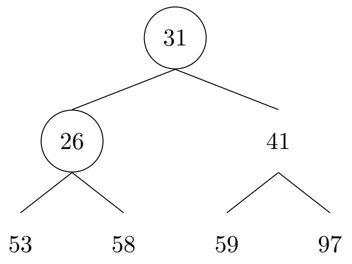
53, 26, 41, 58, 31, 59, 97



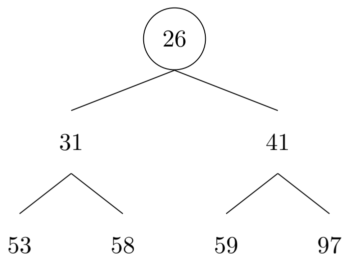
53, 26, 41, 31, 58, 59, 97



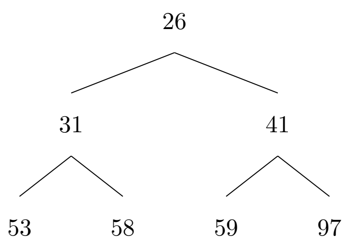
26, 41, 31, 53, 58, 59, 97



26, 31, 41, 53, 58, 59, 97



26, 31, 41, 53, 58, 59, 97



26, 31, 41, 53, 58, 59, 97

Here, we first remove the largest key, which is 97 in this case, and rearrange the heap. Then, we place 97 in the hole left empty due to the deletion. By doing so recursively, we obtain a sorted list.

8.7 Merge Sort

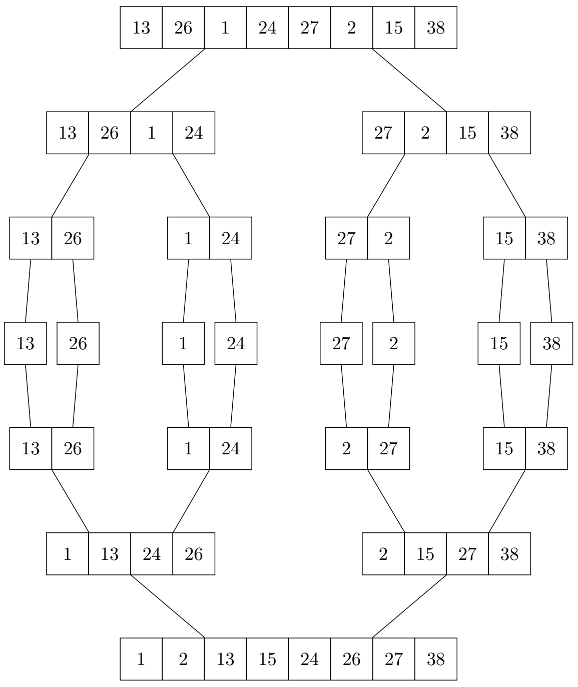
8.7.1 Operation

Merge sort is an excellent method for external sorting, which is used for problems where the data are kept on disks or magnetic tapes, rather than in high-speed memory (RAM).

This algorithm is a classic divide-and-conquer strategy. The problem is divided into smaller sub-problems and solved recursively. The conquering phase consists of patching together the solutions to the sub-problems. Divide-and-conquer is a very powerful use of recursion that we will see many times.

Comparisons of keys are done at only one place in the entire merge sort procedure. This place is within the main loop of the merge procedure. After each comparison, one of the two nodes is sent to the output list. Hence, the number of comparisons certainly cannot exceed the number of nodes being merged.

As shown on the right, we divide the array into two halves recursively until it cannot be further divided. Then, we perform merge sort recursively. For the first merge sort, since there is only one element, it is already sorted. Then, we merge two elements and sort them in order. By doing this recursively, we obtain the sorted list.



Merge Sort

8.7.2 Analysis

It is clear from the tree that the total length of the lists on each level is precisely n, the total number of entries. In other words, every entry is treated in exactly one merge on each level. Hence, the total number of comparisons done on each level cannot exceed n. The number of levels, excluding the leaves for which no merges are done, is $\lceil \log n \rceil$, the ceiling of $\log n$. Therefore, the total number of comparisons of keys done by merge sort on a list of n entries is no more than $n \log n$, rounded up.

Proof.

$$
\begin{aligned}
T(1) &= 1 \\
T(n) &= 2T(\frac{n}{2}) + n \\
\frac{T(n)}{n} &= \frac{T(\frac{n}{2})}{\frac{n}{2}} + 1 \\
\frac{T(\frac{n}{2})}{\frac{n}{2}} &= \frac{T(\frac{n}{4})}{\frac{n}{4}} + 1 \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + 1 \\
\frac{T(n)}{n} &= \frac{T(1)}{1} + \log n \\
T(n) &= n \log n + n = O(n \log n)
\end{aligned}
$$

■

8.8 Quick Sort

8.8.1 Operation

Quicksort is a divide-and-conquer algorithm used for sorting a subarray $A[p \dots r]$. The array is first partitioned (rearranged) into two nonempty subarrays $A[p \dots q]$ and $A[q+1 \dots r]$, such that every element in $A[p \dots q]$ is less than or equal to every element in $A[q+1 \dots r]$. The index q is computed as part of this partitioning procedure.

Then, the two subarrays $A[p \dots q]$ and $A[q+1 \dots r]$ are sorted recursively using quicksort. Since the subarrays are sorted in place, no additional work is required to combine them—thus, the entire array becomes sorted.

R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	m	n
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

In Quicksort, we choose a pivot, where elements to the left of the pivot are smaller, and elements to the right are larger. As shown above, since we choose the first element as the pivot, we first swap it with the middle element. Then, we rearrange the array so that all elements smaller than 26 are placed to the left, and all elements larger than 26 are placed to the right.

8.8.2 Analysis

The number of comparisons of keys that will have been made in the call to `partition` is $n-1$, since every entry in the list is compared to the pivot, except for the pivot entry itself.

Let us denote by $C(n)$ the average number of comparisons done by Quicksort on a list of length n, and by $C(n,p)$ the average number of comparisons on a list of length n where the pivot for the first partition turns out to be the pth smallest element. The remaining work is then $C(p-1)$ and $C(n-p)$ comparisons for the sublists. So, for $n \geq 2$, we have:

$$C(n,p) = (n-1) + C(p-1) + C(n-p)$$

Assuming every pivot is equally likely, we can average this over $p = 1$ to n, giving:

$$C(n) = \frac{1}{n} \sum_{p=1}^{n} [(n-1) + C(p-1) + C(n-p)] = (n-1) + \frac{1}{n} \sum_{p=1}^{n} [C(p-1) + C(n-p)]$$

Since p goes from 1 to n, both $p-1$ and $n-p$ range from 0 to $n-1$. This symmetry gives:

$$C(n) = (n-1) + \frac{2}{n} \sum_{k=0}^{n-1} C(k)$$

The average time is then:

$$C(n) = (n-1) + \frac{2}{n} \sum_{k=0}^{n-1} C(k)$$

For a list of length $n-1$, we have:

$$C(n-1) = (n-2) + \frac{2}{n-1} \sum_{k=0}^{n-2} C(k)$$

Multiplying the first expression by n, and the second by $n-1$, and subtracting them, we obtain:

$$nC(n) - (n-1)C(n-1) = n(n-1) - (n-1)(n-2) + 2C(n-1)$$

which can be rearranged as:

$$\begin{aligned}
\frac{C(n)-2}{n+1} &= \frac{C(n-1)-2}{n} + \frac{2}{n+1} \\
&= \frac{C(n-2)-2}{n-1} + \frac{2}{n+1} + \frac{2}{n} \\
&= \frac{C(n-3)-2}{n-2} + \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} \\
&= \frac{C(2)-2}{3} + \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{4} \\
&= \frac{C(1)-2}{2} + \frac{2}{n+1} + \frac{2}{n} + \frac{2}{n-1} + \cdots + \frac{2}{4} + \frac{2}{3} \\
&= -1 + 2\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{4} + \frac{1}{3}\right)
\end{aligned}$$

We introduce the harmonic numbers $H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$. Approximating this sum using an integral, we evaluate:

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}} \frac{1}{x}\,dx = \log\left(n + \frac{1}{2}\right) - \log\left(\frac{1}{2}\right) \approx \log n + 0.7,$$

which shows that:

$$H(n) = \log n + O(1).$$

By substitution into the recurrence for the average number of comparisons in Quicksort, we find:

$$\frac{C(n)-2}{n+1} = 2\log n + O(1) \quad \Longrightarrow \quad C(n) = 2n\log n + O(n).$$

Thus, in the average case, Quicksort performs $C(n) = 2n\log n + O(n)$ comparisons of keys to sort a list of n entries.

We can choose any entry we wish as the pivot and swap it with the first entry before beginning the loop that partitions the list. The first entry is often a poor choice: if the list is already sorted, then the first key will have no others less than it, and one of the sublists will be empty. Hence, it may be better to select an entry near the center of the list in the hope that the pivot will partition the keys so that approximately half fall on each side.

One method we can use is to select three random elements from the list and choose the median of these three as the pivot. This increases the likelihood that the pivot will divide the list more evenly, ensuring that neither sublist is empty. By selecting more elements and choosing the median among them, the quality of the pivot can be further improved, reducing the chance of highly unbalanced partitions.

8.9 Radix Sort

This is also called **bucket sort** or **postman sort**.

In some special cases, sorting can be performed in linear time. The idea is to consider the key one character at a time and to divide the items not into two sublists, but into as many sublists as there are possible values for the current character. For example, if the keys are alphabetic strings, we divide the list into 26 sublists at each stage—one for each letter of the alphabet. That is, we set up a table of 26 lists and distribute the items into these lists according to one of the characters in their keys.

The time used by radix sort is proportional to nk, where n is the number of items being sorted, and k is the number of characters in a key. The time for other sorting methods depends on n but not directly on the length of the key. For instance, the best time among comparison-based sorts was achieved by mergesort, which takes $n\log n + O(n)$ time.

The relative performance of the sorting methods therefore depends on the relationship between nk and $n \log n$. If the keys are long (large k) but there are relatively few of them (small n), then comparison-based methods such as mergesort will likely outperform radix sort.

However, if k is small and there are many keys (large n), then radix sort can be faster than any of the comparison-based methods we have studied.

8.10 More on Sorting

When sorting large structures, it is often impractical to store all the information of each record directly in an array. This is because such records can be large, making data movement expensive and swapping inefficient.

A practical solution is to let the input array contain **pointers** to the actual structures rather than the structures themselves. We can then perform the sort by comparing the keys that the pointers reference and swapping the pointers when necessary. In this way, all data movement is essentially the same as if we were sorting integers.

This technique is known as **indirect sorting**, or **sorting by address**. It can be applied to most of the data structures we have described and is especially useful when dealing with large datasets.

When performing sorting, we often care about **stability**. A sorting function is called **stable** if, whenever two items have equal keys, they remain in the same relative order in the output as they were in the input.

Stability is especially important when a list has already been sorted by one key and is now being sorted by another key. In such cases, preserving the original ordering as much as possible is desirable.

Formally, an algorithm is stable if for all records i and j such that $k[i] = k[j]$, if $r[i]$ precedes $r[j]$ in the original file, then $r[i]$ also precedes $r[j]$ in the sorted file. That is, a stable sort keeps records with the same key in the same relative order as before the sort.

For example, consider the input list:

$$(a, 1),\ (b, 2),\ (c, 3),\ (a, 4),\ (a, 5),\ (b, 6),\ (c, 7)$$

A stable sort on the first element yields:

$$(a, 1),\ (a, 4),\ (a, 5),\ (b, 2),\ (b, 6),\ (c, 3),\ (c, 7)$$

However, if we use an unstable algorithm such as insertion sort (in a naive implementation), we might get:

$$(a, 5),\ (a, 4),\ (a, 1),\ (b, 6),\ (b, 2),\ (c, 7),\ (c, 3)$$

which is not stable.

> **Remark.** By definition, Shell sort is not stable.

Chapter 9

Graph Algorithm

9.1 Definitions

A **graph** is an important mathematical structure. A graph $G = (V, E)$ consists of a set of vertices (or nodes) V and a set of edges E. Each edge is a pair (v, w), where $v, w \in V$. Edges are sometimes referred to as *arcs*.

If $e = (v, w)$ is an edge with vertices v and w, then v and w are said to *lie on* e, and e is said to be *incident with* v and w.

If the pairs are unordered, then G is called an **undirected graph**. If the pairs are ordered, then G is called a **directed graph**. The term *directed graph* is often shortened to *digraph*, and the unqualified term *graph* usually refers to an undirected graph.

Vertex w is said to be *adjacent to* vertex v if and only if $(v, w) \in E$. In an undirected graph, if $(v, w) \in E$, then $(w, v) \in E$ as well, so v is adjacent to w and w is adjacent to v. Sometimes, an edge has a third component, known as either a *weight* or a *cost*.

A **path** in a graph is a sequence of vertices $w_1, w_2, \dots, w_n$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq n - 1$. The *length* of such a path is the number of edges on the path, which is equal to $n - 1$.

We allow a path from a vertex to itself. If this path contains no edges, then the path length is 0. If the graph contains an edge (v, v) from a vertex to itself, then the path v, v is sometimes referred to as a **loop**.

A **simple path** is a path in which all vertices are distinct, except that the first and last vertices may be the same.

A cycle in a directed graph is a path of length at least 1 such that $w_1 = w_n$. This cycle is called *simple* if the path is simple. For undirected graphs, we require that the edges be distinct. The logic behind this requirement is that the path u, v, u in an undirected graph should not be considered a cycle, because (u, v) and (v, u) represent the same edge. However, in a directed graph, these are different edges, so it makes sense to call this a cycle.

A directed graph is acyclic if it has no cycles. A directed acyclic graph is sometimes referred to by its abbreviation, DAG.

An undirected graph is connected if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected. If a directed graph is not strongly connected, but the underlying graph (without direction to the arcs) is connected, then the graph is said to be weakly connected. A complete graph is a graph in which there is an edge between every pair of vertices.

9.2 Implementation

Consider each airport as a vertex, and two vertices are connected by an edge if there is a nonstop flight between the airports represented by the vertices. The edge could have a weight, representing the time, distance, or cost of the flight. Such a graph is directed, since it might take longer or cost more to fly in different directions.

We would like to make sure that the airport system is strongly connected, so that it is always possible to fly from any airport to any other airport. We might also like to quickly determine the best flight between any two airports. This could mean the path with the fewest number of edges or could be taken with respect to one or all of the weight measures.

To implement such a system, we can use a **two-dimensional array**. This is known as an **adjacency matrix representation**. For each **edge** (u, v), we set $a[u][v] = 1$, otherwise the entry in the array is 0. If the edge has a **weight** associated with it, then we can set $a[u][v]$ equal to the weight and use either a very large or a very small weight as a **sentinel** to indicate **nonexistent edges**.

Then, if we were looking for the cheapest airplane route, we could represent nonexistent flights with a cost of ∞. If we were somehow looking for the most expensive airplane route, we could use $-\infty$ to represent nonexistent edges.

The space requirement is $\Theta(|V|^2)$. This is unacceptable if the graph does not have many edges. An adjacency matrix is an appropriate representation if the graph is dense, $|E| = \Theta(|V|^2)$.

However, if the graph is not dense but sparse, a better solution is an adjacency list representation. For each vertex, we keep a list of all adjacent vertices. The space requirement is then $O(|E| + |V|)$. If the edges have weights, then this additional information is also stored in the cells.

9.3 Topological Sort

A **topological sort** is a **linear ordering of vertices** in a **directed acyclic graph (DAG)** such that for every directed edge from vertex u to vertex v, vertex u comes before vertex v in the ordering. A directed edge (v, w) indicates that course v must be completed before course w may be attempted.

A topological ordering is then a linear sequence of the vertices such that for every directed edge from vertex u to vertex v, u appears before v in the sequence. In other words, the ordering respects all the directed dependencies in the graph.

Notice that a topological ordering is not possible with a cyclic graph, since for two vertices v and w on the cycle, v precedes w and w precedes v. The ordering is not necessarily unique; any valid ordering will suffice.

To find the topological ordering, we first define the **in-degree** of a vertex v as the number of incoming edges (u, v). We compute the in-degrees of all vertices in the graph. Then, we find any vertex with an in-degree of zero, print this vertex, and remove it along with its outgoing edges from the graph. We then apply this same strategy to the rest of the graph.

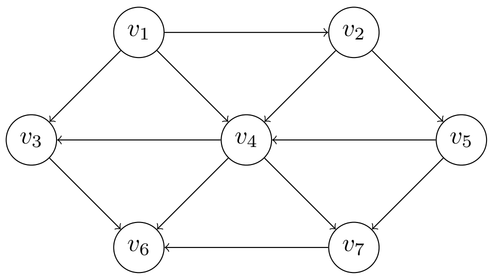


Figure 9.1: Topological Sort

	In-degree before dequeue						
Vertex	1	2	3	4	5	6	7
v_1	0	0	0	0	0	0	0
v_2	1	0	0	0	0	0	0
v_3	2	1	1	1	0	0	0
v_4	3	2	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	3	3	3	3	2	1	0
v_7	2	2	2	1	0	0	0
`enqueue`	v_1	v_2	v_5	v_4	v_3	v_7	v_6
`dequeue`	v_1	v_2	v_5	v_4	v_3	v_7	v_6

Table 9.1: Topological Sorting

Since it is a simple sequential scan of the in-degree array, each call to it takes $O(|V|)$ time. Since there are $|V|$ such calls, the running time of the algorithm is $O(|V|^2)$.

9.4 Algorithms

Here we introduce some algorithms that are related to graphs.

9.4.1 Shortest Path Algorithm

For the shortest path algorithm, the input is a weighted graph. Associated with each edge (v_i, v_j) is a cost $c_{i,j}$ to traverse the arc. The cost of a path $v_1, v_2, \cdots, v_n$ is $\sum_{i=1}^{n-1} c_{i,i+1}$. This is referred to as the weighted path length. The unweighted path length is merely the number of edges on the path, namely $n - 1$.

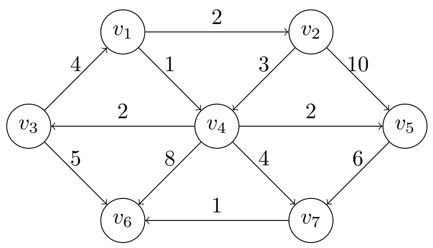


Figure 9.2: A weighted graph

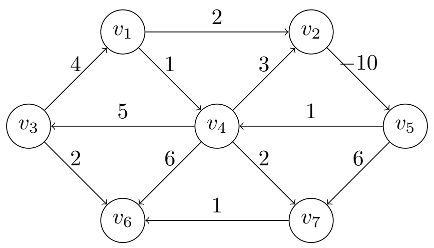


Figure 9.3: A weighted graph

For example, given as input a weighted graph $G = (V, E)$ and a distinguished vertex s, we are asked to find the shortest weighted path from s to every other vertex in G.

In Figure 9.2, the shortest weighted path from v_1 to v_6 has a cost of 6 and goes from v_1 to v_4 to v_7 to v_6. The shortest unweighted path between these vertices has a length of 2.

There could also be cases where the cost is negative. For example, in Figure 9.3, the path from v_5 to v_4 has cost 1, but a shorter path exists by following the loop v_5, v_4, v_2, v_5, v_4, which has a cost of -5. The shortest path between v_5 and v_4 is undefined. This loop is known as a negative-cost cycle. When one is present in the graph, the shortest paths are not defined.

Currently, there are no algorithms in which finding the path from s to one vertex is significantly faster (by more than a constant factor) than finding the paths from s to all vertices. The intermediate nodes in a shortest path must also lie on the shortest paths from s.

If we assume that there are no negative edges, then for the weighted shortest path problem, the running time of the algorithm will be $O(|E| \log |V|)$ when implemented with reasonable data structures.

If the graph has negative edges, we will have a poor time bound of $O(|E| \cdot |V|)$. We will solve the weighted problem for the special case of acyclic graphs in linear time.

9.4.2 Breadth-First Search

We want to find the unweighted shortest path. Using vertex s, we would like to find the shortest path from s to all other vertices. There are no weights on the edges, which is a special case of the weighted shortest-path problem, since we could assign all edges a weight of 1.

We can use **breadth-first search (BFS)** to search for the shortest path. It operates by processing vertices in layers. The vertices closest to the start are evaluated first, and the most distant vertices are evaluated last. This is much the same as a level-order traversal for trees.

For each vertex, we will keep track of three pieces of information. First, we will keep its distance from s in the entry d_v. Initially, all vertices are unreachable except for s, whose path length is 0. The entry in p_v is the bookkeeping variable, which will allow us to print the actual paths. The entry `known` is set to 1 after a vertex is processed. Initially, all entries are unknown, including the start vertex. Once it is known, we have a guarantee that no cheaper path will ever be found, and so processing for that vertex is essentially complete.

In Figure 9.4, suppose we choose s to be v_3. The shortest path from s to v_3 is then a path of length 0. Now we can start looking for all vertices that are a distance 1 away from s. These can be found by looking at the vertices that are adjacent to s. Then we have

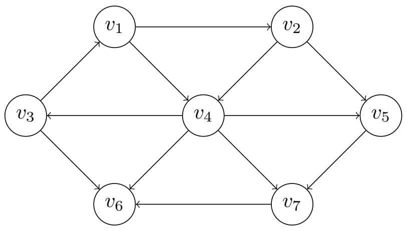


Figure 9.4: Unweighted graph

v	Known	d_v	p_v
v_1	0	∞	0
v_2	0	∞	0
v_3	0	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0
Q	v_3		

Initial State

v	Known	d_v	p_v
v_1	0	1	v_3
v_2	0	∞	0
v_3	1	0	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	1	v_3
v_7	0	∞	0
Q	v_1, v_6		

v_3 dequeued

v	Known	d_v	p_v
v_1	1	1	v_3
v_2	0	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	∞	0
v_6	0	1	v_3
v_7	0	∞	0
Q	v_6, v_2, v_4		

v_1 dequeued

v	Known	d_v	p_v
v_1	1	1	v_3
v_2	0	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	∞	0
v_6	1	1	v_3
v_7	0	∞	0
Q	v_2, v_4		

v_6 dequeued

v	Known	d_v	p_v
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	0	2	v_1
v_5	0	3	v_2
v_6	1	1	v_3
v_7	0	∞	0
Q	v_4, v_5		

v_2 dequeued

v	Known	d_v	p_v
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	0	3	v_2
v_6	1	1	v_3
v_7	0	3	v_4
Q	v_5, v_7		

v_4 dequeued

v	Known	d_v	p_v
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	0	3	v_4
Q	v_7		

v_5 dequeued

v	Known	d_v	p_v
v_1	1	1	v_3
v_2	1	2	v_1
v_3	1	0	0
v_4	1	2	v_1
v_5	1	3	v_2
v_6	1	1	v_3
v_7	1	3	v_4
Q			

v_7 dequeued

The running time for this algorithm is $O(|V|^2)$, because of the doubly nested `for` loops.

9.4.3 Dijkstra's Algorithm

If the graph is weighted, the problem becomes harder. However, we can still use the idea from the unweighted case.

Again, each vertex is marked as either known or unknown. A tentative distance d_v is kept for each vertex, as before. This distance turns out to be the shortest path length from s to v using only known vertices as intermediates. As before, we record p_v, which is the last vertex to cause a change to d_v.

The general method to solve the single-source shortest-path problem is known as Dijkstra's algorithm. This is a prime example of a greedy algorithm. Greedy algorithms generally solve a problem in stages by doing what appears to be the best choice at each stage.

Notice that Dijkstra's algorithm proceeds in stages. At each stage, Dijkstra's algorithm selects a vertex v, which has the smallest d_v among all the unknown vertices, and declares that the shortest path from

s to v is known. The remainder of a stage consists of updating the values of d_w.

Consider the following example:

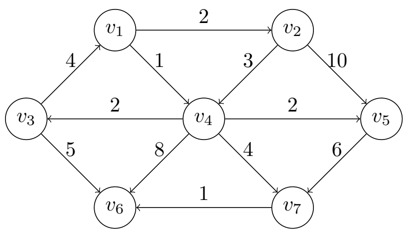


Figure 9.5: A weighted graph

v	Known	d_v	p_v
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Initial State

v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	∞	0
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

After v_1

v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

After v_4

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

After v_2

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	8	v_3
v_7	0	5	v_4

After v_3 and v_5

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	6	v_7
v_7	1	5	v_4

After v_7

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	1	6	v_7
v_7	1	5	v_4

After v_6

However, Dijkstra's algorithm does not work with negative edge costs. A combination of the weighted and unweighted algorithms will solve the problem, but at the cost of a drastic increase in running time. The running time is $O(|E| \cdot |V|)$ if adjacency lists are used.

We can improve this algorithm by changing the order in which vertices are declared known, otherwise known as the vertex selection rule. The new rule is to select vertices in topological order. The algorithm can be done in one pass, since the selections and updates can take place as the topological sort is being performed.

The selection rule works because when a vertex v is selected, its distance d_v can no longer be lowered. Since, by the topological ordering rule, it has no incoming edges emanating from unknown nodes, the running time is $O(|E| + |V|)$, since the selection takes constant time.

There are several applications, including the downhill skiing problem, modeling of chemical reactions, and critical path analysis. For critical path analysis, each node represents an activity that must be performed, along with the time it takes to complete the activity. This graph is thus known as an activity-node graph.

For an activity-node graph, the edges represent precedence relationships. An edge (v, w) means that activity v must be completed before activity w may begin. This implies that the graph must be acyclic.

If we need to find the shortest paths between all pairs of vertices in the graph, a brute-force method is to run the appropriate single-source algorithm $|V|$ times. On sparse graphs, it is faster to run $|V|$ Dijkstra's algorithms coded with priority queues.

9.5 Maximum-Flow Algorithm

Suppose we are given a directed graph $G = (V, E)$ with edge capacities $c_{v,w}$. These capacities could represent the amount of water that can flow through a pipe or the amount of traffic that can flow on a street between two intersections. We have two vertices: s, which we call the source, and t, which is the sink. Through any edge (v, w), at most $c_{v,w}$ units of "flow" may pass.

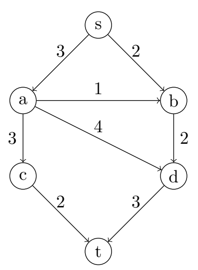
For any vertex v that is neither s nor t, the total flow coming in must equal the total flow going out. The maximum flow problem is to determine the maximum amount of flow that can pass from s to t.

To solve this problem, we can use a simple maximum-flow algorithm. We have G_f as a flow graph, which represents the flow that has been attained at any stage in the algorithm. Initially, all edges in G_f have no flow. G_f should contain a maximum flow when the algorithm terminates.

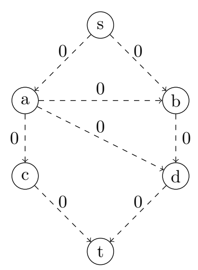
We also have G_r, the residual graph. G_r tells, for each edge, how much more flow can be added. We calculate this by subtracting the current flow from the capacity for each edge. An edge in G_r is known as a residual edge.

At each stage, we find a path in G_r from s to t. This path is known as an augmenting path. The minimum edge on this path determines the amount of flow that can be added to every edge on the path. We do this by adjusting G_f and recomputing G_r. When we find no path from s to t in G_r, we terminate. This algorithm is nondeterministic in that we are free to choose any path from s to t.

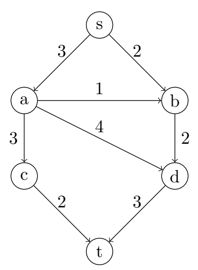
Consider the following example:



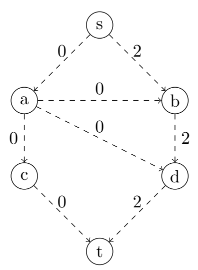
Initial Stage G



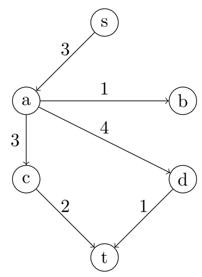
1 Flow Graph G_f



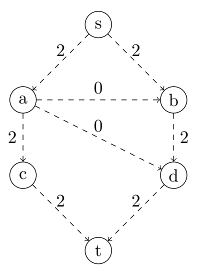
1 Residual Graph G_r



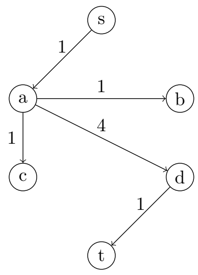
2 Flow Graph G_f



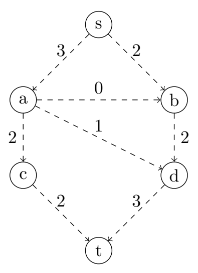
2 Residual Graph G_r



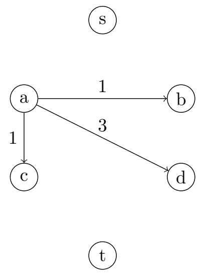
3 Flow Graph G_f



3 Residual Graph G_r



4 Flow Graph G_f



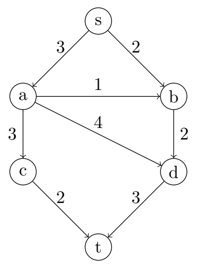
4 Residual Graph G_r

Starting from the initial stage, we add different units of flow according to the bottleneck. At the end, there is no more path from s to t, and thus the algorithm terminates.

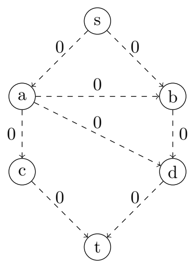
The resulting flow of 5 happens to be the maximum flow. However, this is not the optimal solution. Suppose we choose the path s, a, d, t as the initial path. Then, the result of this choice is that there is no longer any path from s to t in the residual graph.

We can optimize it by allowing the algorithm to change its mind. To do this, for every edge (v, w) with flow $f_{v,w}$ in the flow graph, we will add an edge in the residual graph (w, v) with capacity $f_{v,w}$. In effect, we are allowing the algorithm to undo its decisions by sending flow back in the opposite direction.

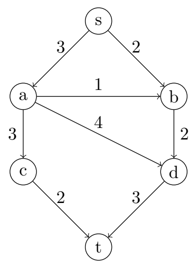
Consider the following example follows the same initial stage:



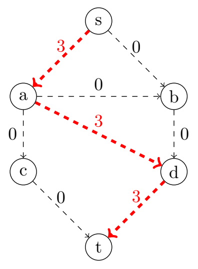
Initial Stage G



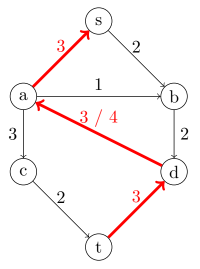
1 Flow Graph G_f



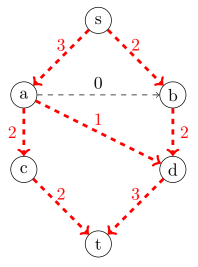
1 Residual Graph G_r



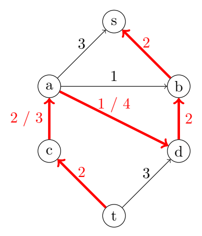
2 Flow Graph G_f



2 Residual Graph G_r



3 Flow Graph G_f



3 Residual Graph G_r

As seen from this flow, although we initially choose the path s, a, d, t, in the residual graph, there are edges in both directions between a and d. Either one more unit of flow can be pushed from a to d, or up to three units can be pushed back—thus, we can undo flow. Now, the algorithm finds the augmenting path s, b, d, a, c, t. By pushing two units of flow from d to a, the algorithm removes two units of flow from the edge (a, d), effectively changing its previous decision.

Notice that if the capacities are all integers and the maximum flow is f, then since each augmenting path increases the flow value by at least 1, f stages suffice. The total running time is $O(f \cdot |E|)$, since an augmenting path can be found in $O(|E|)$ time using an unweighted shortest-path algorithm.

Remark. We always choose the augmenting path that allows the largest increase in the flow. Finding such a path is similar to solving a weighted shortest-path problem.

9.6 Minimum Spanning Tree

Informally, a *minimum spanning tree* of an undirected graph G is a tree formed from graph edges that connects all the vertices of G at the lowest total cost. A minimum spanning tree exists if and only if G is connected.

The minimum spanning tree is unique only when all edge weights are distinct.

Note that the number of edges in a minimum spanning tree is $|V| - 1$. It is a *tree* because it is acyclic, *spanning* because it covers all the vertices, and *minimum* because the sum of the edge costs is the lowest possible.

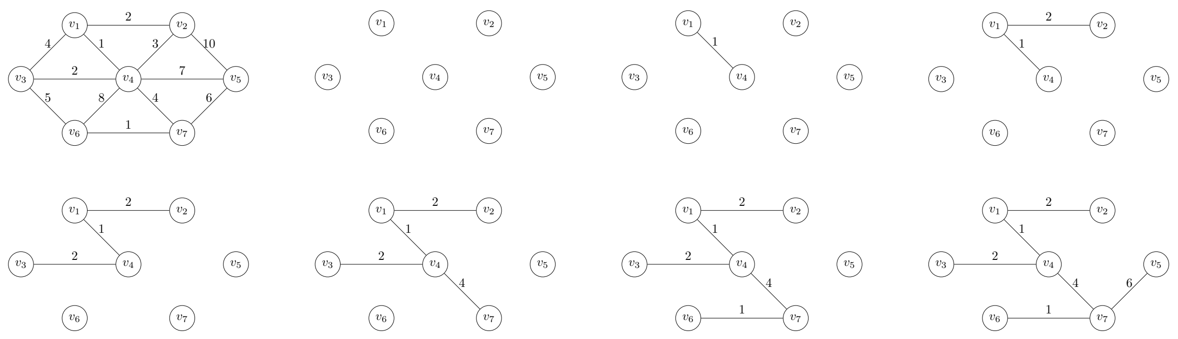
We can construct a minimum spanning tree using **Prim's algorithm**, which grows the tree in successive stages. At each stage, starting from a root node, we add an edge—and thus an associated vertex—to the tree. The algorithm selects the edge (u, v) with the smallest cost such that u is in the tree and v is not.

Prim's algorithm is essentially identical to Dijkstra's algorithm for shortest paths. For each vertex, we maintain values d_v and p_v, as well as an indication of whether it is *known* or *unknown*.

Here, d_v is the weight of the shortest edge connecting v to any known vertex. The variable p_v, as before, records the last vertex to cause a change in d_v. After a vertex v is selected, for each unknown neighbor w, we update:

$$d_w = \min(d_w, c_{w,v})$$

Consider the following example:



We can also use tables to show the results:

v	Known	d_v	p_v
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

Initial Stage

v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	4	v_1
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

After v_1

v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	8	v_4
v_7	0	4	v_4

After v_4

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	7	v_4
v_6	0	5	v_3
v_7	0	4	v_4

After v_2 and v_3

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	0	6	v_7
v_6	0	1	v_7
v_7	1	4	v_4

After v_7

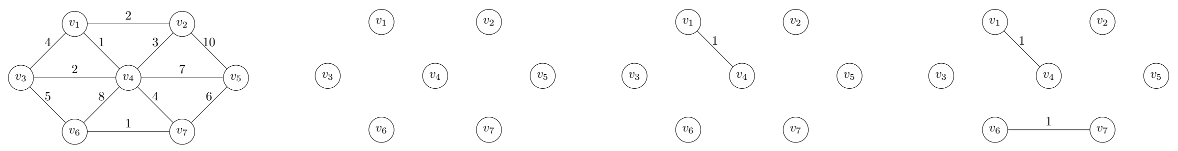
v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	2	v_4
v_4	1	1	v_1
v_5	1	6	v_7
v_6	1	1	v_7
v_7	1	4	v_4

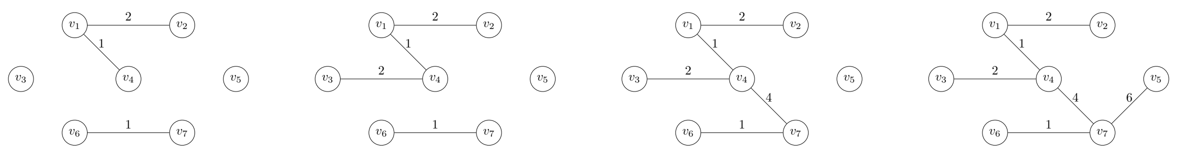
After v_6

Be aware that Prim's algorithm runs on *undirected* graphs, so when implementing it, remember to insert every edge into *both* adjacency lists.

The running time of Prim's algorithm is $O(|V|^2)$ when implemented without heaps, which is optimal for dense graphs. With a binary heap, the running time improves to $O(|E|\log|V|)$, which is more efficient for sparse graphs.

A second greedy strategy for constructing a minimum spanning tree is to continually select edges in order of increasing weight, accepting an edge only if it does not form a cycle. This is called the **Kruskal's Algorithm**. This method maintains a *forest*—a collection of disjoint trees. Adding an edge merges two trees into one. When the algorithm terminates, the forest has been reduced to a single tree, which is the minimum spanning tree. Consider the following example:





This algorithm terminates when enough edges have been accepted to form a spanning tree. It is simple to decide whether an edge (u, v) should be accepted or rejected. The appropriate data structure for making this decision is the *union-find* (or disjoint-set) algorithm. The key invariant is that, at any point in the process, two vertices belong to the same set if and only if they are connected in the current spanning forest.

Initially, each vertex is placed in its own set. If u and v are found to be in the same set, the edge (u, v) is rejected, since adding it would form a cycle. Otherwise, the edge is accepted, and a union operation is performed on the two sets containing u and v.

The worst-case running time of this algorithm is $O(|E| \log |E|)$, dominated by the heap operations required for edge sorting. Since $|E| = O(|V|^2)$, this time complexity simplifies to $O(|E| \log |V|)$. In practice, the algorithm performs significantly faster than this theoretical bound might suggest.

9.7 Depth-First Search

Depth-First Search (DFS) is a generalization of pre-order traversal. Starting at some vertex v, we process v and then recursively traverse all vertices adjacent to v. If this process is performed on a tree, then all tree vertices are systematically visited in a total of $O(|E|)$ time.

If we perform this process on an arbitrary graph, we need to be careful to avoid cycles. To do this, when we visit a vertex v, we mark it as visited to indicate that we have already been there, and then recursively call depth-first search on all adjacent vertices that are not already marked.