UMBC
Training Centers

**6996 Columbia Gateway Drive
Suite 100
Columbia, MD 21045
Tel: 443-692-6600
http://www.umbctraining.com**

DATA STRUCTURES & ALGORITHMS I

Course # TCPRG4001

Rev. 12/28/2020

This page intentionally left blank.

Contents

©2017 UMBC Training Centers

©2017 UMBC Training Centers

©2017 UMBC Training Centers

©2017 UMBC Training Centers

©2017 UMBC Training Centers

This page intentionally left blank.

©2017 UMBC Training Centers

Chapter 1

Asymptotic Limit

Asymptotic

The limit at which a function is arbitrarily close.

It is used to describe the limiting behavior of a function as the input approaches some value.

Meaning the value a function approaches as the input approaches some value.

In asymptotic analysis of algorithms the limits on input are typically at the extremes of $\pm\infty$ or some sufficiently small or large value.

A function is said to be asymptotically equivalent to another function if the two have the same limiting behavior.

For example, consider a polynomial function of n. The value that the function approaches as n goes to infinity is dominated by its highest degree monomial. The polynomial is therefore asymptotically equivalent to its highest degree monomial.

©2017 UMBC Training Centers

Description

The asymptotic limits are used in the analysis of data structures and algorithms to understand the best and worst case resource requirements.

The upper bound on performance describes the worst-case runtime of an algorithm, meaning as the input goes to infinity the performance is no worse than the limit of the function.

The lower bound gives the best-case runtime, or minimum number of steps required to complete.

The asymptotic analysis of the time and space complexity of algorithms and data structures is therefore concerned with the asymptotic growth of functions describing the time and space limits.

Limits

The limit of a function is the value it approaches as the input goes to some value.

For example, the limit of $f(x) = x^3 + x - 6$ as x goes to 3 means $f(x)$ approaches the value of 24. This is written as,

$$\lim_{x \to 3}(x^3 + x - 6) = 3^3 + 3 - 6 = 24.$$

Suppose now that $x \to \infty$. Then the value of $f(x)$ also approaches ∞. But importantly we have the equality,

$$\lim_{x \to \infty}(x^3 + x - 6) = \lim_{x \to \infty} x^3.$$

This can be demonstrated as follows.

$$\begin{aligned}
\lim_{x \to \infty}(x^3 + x - 6) &= \lim_{x \to \infty} x^3\Big(1 + \frac{1}{x^2} - \frac{6}{x^3}\Big) \\
&= \lim_{x \to \infty} x^3\Big(1 + 0 - 0\Big) \\
&= \lim_{x \to \infty} x^3
\end{aligned}$$

By factoring out the largest degree term, all other terms are now divided by some power of x. Since a constant divided by an increasingly large number tends to zero, then $\lim_{x\to\infty}\frac{1}{x^k} = 0$ for $k > 0$.

Thus the limiting behavior of $f(x) = x^3 + x - 6$ is the same as x^3 as x goes to infinity.

Hence it can be shown in the same manner that the limit of a polynomial is equivalent to the limit of its highest degree monomial.

$$
\begin{aligned}
&\lim_{x\to\infty} c_n x^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \ldots + c_0 \\
&= \lim_{x\to\infty} x^n \Big(c_n + \tfrac{c_{n-1}}{x^1} + \tfrac{c_{n-2}}{x^2} + \ldots + \tfrac{c_0}{x^n} \Big) \\
&= \Big(\lim_{x\to\infty} x^n \Big) \Big(\lim_{x\to\infty} (c_n + \tfrac{c_{n-1}}{x^1} + \tfrac{c_{n-2}}{x^2} + \ldots + \tfrac{c_0}{x^n}) \Big) \\
&= \Big(\lim_{x\to\infty} x^n \Big) (c_n) \\
&= \lim_{x\to\infty} c_n x^n
\end{aligned}
$$

©2017 UMBC Training Centers

Consider the function $f(x) = \frac{x^2-4}{x-2}$.

What is the limit of $f(x)$ as $x \to 4$? We can simply substitute the value of $x = 4$.

$$\lim_{x \to 4} \frac{x^2 - 4}{x - 2} = \frac{16 - 4}{4 - 2} = 6$$

But what is the limit of $f(x)$ as $x \to 2$? Substituting in $x = 2$ yields $\frac{0}{0}$.

This does not mean that the limit of the function as x goes to 2 is indeterminate. Try instead to factor and then substitute.

$$\lim_{x \to 2} \frac{x^2 - 4}{x - 2} = \lim_{x \to 2} \frac{(x-2)(x+2)}{(x-2)} = \lim_{x \to 2} (x+2) = 4.$$

Given other functions, factoring and substitution does not eliminate indeterminate forms such as $\frac{0}{0}$, $\frac{\infty}{\infty}$.

What is $\lim_{x \to \infty} \frac{e^x}{x}$?

For these, we turn to L'Hospital's Rule. This is beyond our scope here, but we invite the reader learn more about it.

Time Complexity

The time complexity of an algorithm refers to the asymptotic limit of its runtime performance.

A function describing algorithm performance gives the growth in the number of steps it takes for the algorithm to complete as the input size increases. This is also known as the work performed by the algorithm.

Let $f(n)$ describe the runtime for some algorithm, where n is a natural number denoting the input size.

If $f(n) = n$ then the number of steps to completion increases at the same rate as the input size; thus the algorithm performance is linear-time.

If instead $f(n) = n^2$, then the number of steps to completion increases at the square rate of the input size; thus the algorithm performance is quadratic-time.

The asymptotic limits of this function gives the bounds on the performance of the algorithm as the input goes to infinity.

This is important because an algorithm may perform very well on small input, but as the input increases in size the performance can degrade significantly.

It is possible that $f(n) \approx n$ for small values of n. But if the limit of $f(n)$ as n goes to infinity behaves like $f(n) = n^2$, then it is useful to know that the runtime cannot be worse than quadratic-time no matter the input size.

Conversely if the work is at least $f(n) = n$, then the algorithm cannot do better than linear-time.

Hence the asymptotic limits guarantee the runtime cannot be any better than the lower limit of the function and no worse than the upper limit.

Asymptotic Growth

Given a function that describes the runtime of an algorithm, then the performance of the algorithm is bounded by the limit of this function as the input approaches infinity.

A function $f(n)$ for the runtime of an algorithm describes the growth in the number of steps it takes for the algorithm to complete with respect to the input size n.

Figure1.1 illustrates the asymptotic growth of common functions for practical runtimes of many algorithms.

The functions with the steepest curves represent slower algorithms because the number of steps to completion grows rapidly with the input size.

The functions in Figure 1.1 listed in order of ascending rate of growth are $\log n$, n, $n \log n$, n^2, and are known as logarithmic, linear, loglinear, and quadratic functions, respectively.

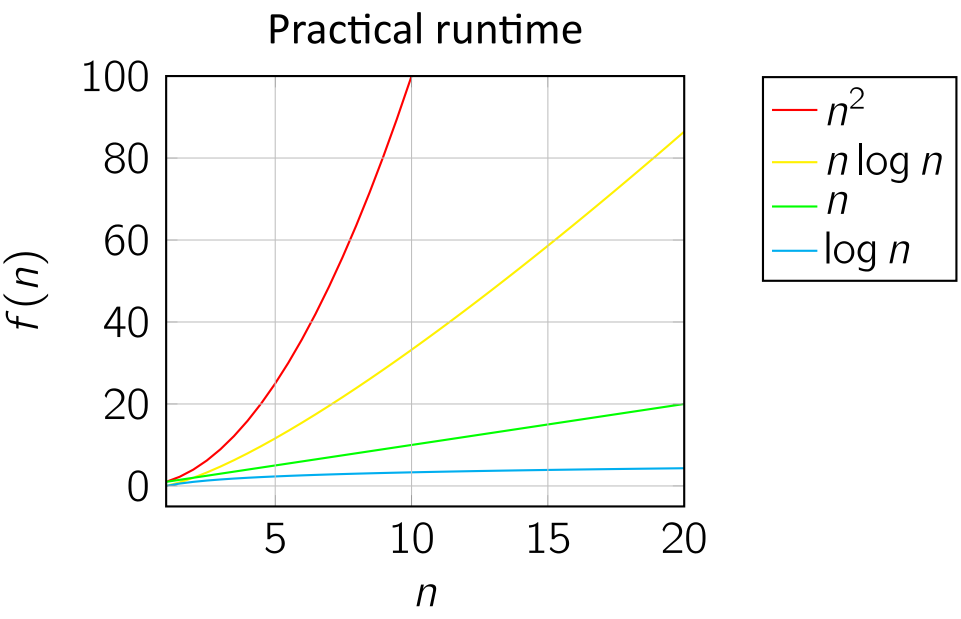


Figure 1.1: Asymptotic growth.

Intractable growth

The next functions plotted in Figure 1.2 are some of the fastest growing functions encountered in the analysis of algorithm performance.

These functions in order of ascending growth are 2^n, $n!$, n^n, and represent exponential, factorial, and superexponential functions, respectively.

It isn't difficult to see the order of growth by simply expanding these functions.

©2017 UMBC Training Centers

$$
\begin{aligned}
2^n &= 2 \cdot 2 \cdot 2 \cdots 2\\
n! &= n \cdot (n-1) \cdot (n-2) \cdots 1\\
n^n &= n \cdot n \cdot n \cdots n
\end{aligned}
$$

An algorithm with runtime that follows any of these functions would be intractable because the number of steps needed for completion grows too rapidly for even moderate input sizes.

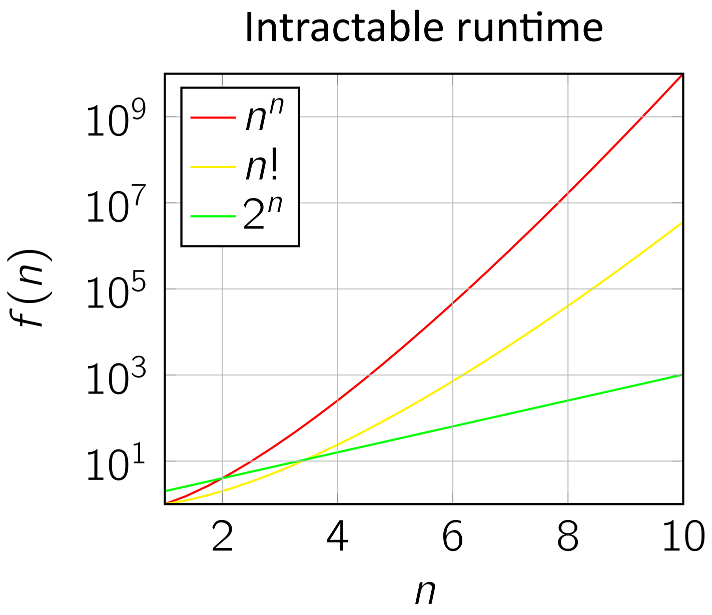


Figure 1.2: Intractable growth.

Logarithmic growth

Some of the slowest growing functions are logarithmic functions. Figure 1.3 illustrates logarithmic growth.

©2017 UMBC Training Centers

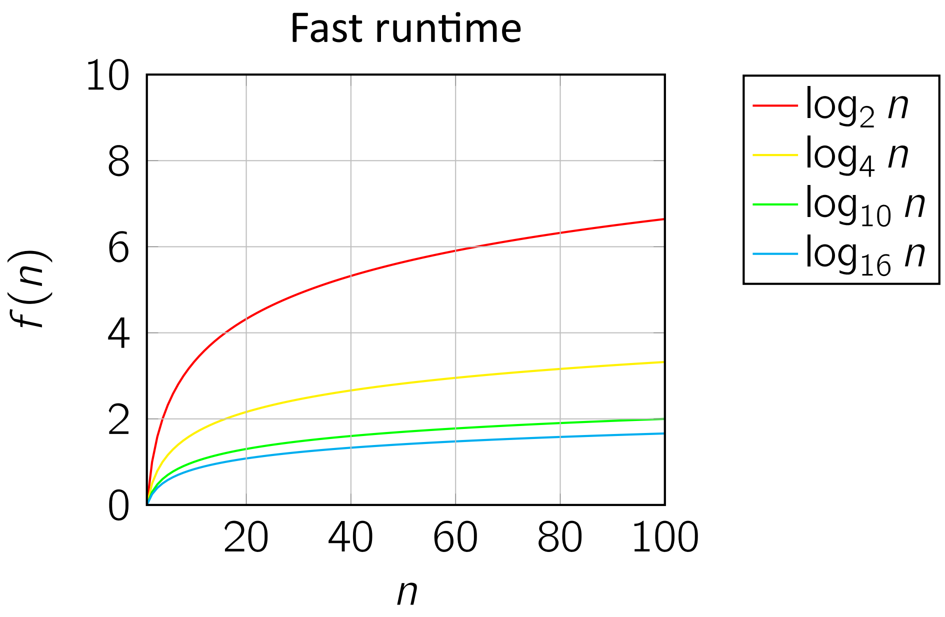


Figure 1.3: Logarithmic growth.

Observe that smaller log bases have faster growth which translates to a slower algorithm runtime.

This is easily demonstrated. Consider the number 1000. In log-base 10 it takes just three steps from 10 to get 1000 by powers of 10. But in log-base 2, it takes over 9 steps ($2^{10} = 1024$).

But asymptotically, the base does not matter!

We can convert from one base to another base by the following. Let's begin with $n = a^x = b^y$, $a = b^{y/x}$. Taking the log of the corresponding bases leads to,

$$\begin{aligned}\log_a n &= x,\\ \log_b n &= y,\\ \log_b a &= y/x.\end{aligned}$$

Then by substitution,

$$\begin{aligned}\log_b a &= y/x\\ &= \frac{\log_b n}{\log_a n}.\end{aligned}$$

Thus we can convert logarithms of different bases using,

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

But notice that the denominator of the right side is a constant if the log bases are constant. Then $\log_a n = c \log_b n$ where $c = \frac{1}{\log_b a}$ is a constant.

This means that $\log_a n$ is within a constant factor of $\log_b n$, hence asymptotically the logarithms are equivalent.

We can see this in Figure 1.4 as the input size n approaches some arbitrarily large value that the values of $f(n)$ for each logarithm converges arbitrarily close.

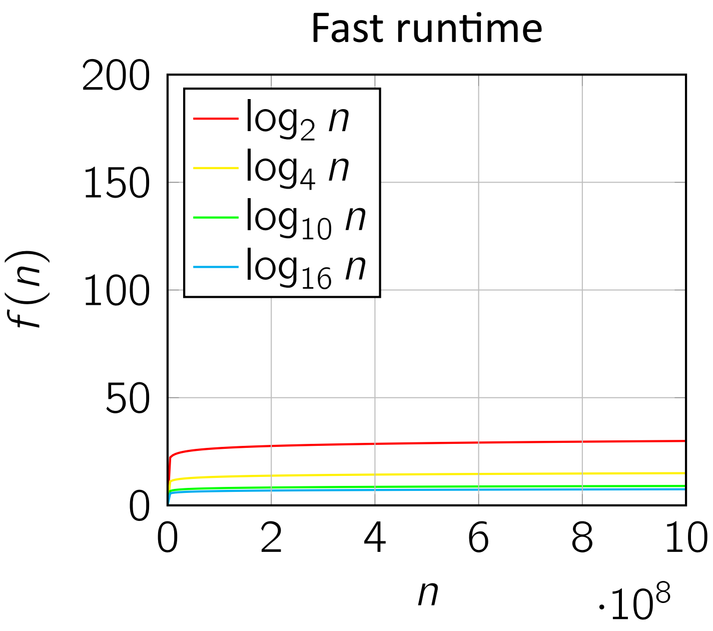


Figure 1.4: Logarithmic growth.

We will use $\log n$ throughout for $\log_2 n$, thus dropping the log base.

Comparing function growth

It is important to know how to identify which functions grow faster than others.

There are a number of analytical methods.

Generally, it is easier to compare functions of the same bases, especially when encountering log exponents.

Compare n^5 and $2^{4 \log n}$. At first glance it may be tempting to conclude that $2^{4 \log n}$ grows faster than the polynomial n^5 because it appears to be an exponential function.

Let's get $2^{4 \log n}$ to base n.

$$\begin{aligned} 2^{4 \log n} &= \left(2^{\log n}\right)^4 \\ &= n^4 \end{aligned}$$

Now it is clear that $2^{4 \log n} < n^5$.

Let's compare 2^n and n^{100}, which are exponential and polynomial functions, respectively.

Suppose $n = 10$, then $2^{10} = 1024$ is clearly much smaller than 10^{100}. But we want to compare these functions in the limit as n goes to infinity. Converting to the same base will help in the analysis.

©2017 UMBC Training Centers

Since $2^{\log n} = n$, then $n^{100} = 2^{100 \log n}$. Now it is a matter of comparing the exponents n and $100 \log n$.

For small values of n the large constant factor of 100 pushes the $\log n$ function faster (e.g. n=16). But suppose instead $n = 2^{32}$, then clearly $100 \log 2^{32} = 3200$ is much smaller than 2^{32}.

Thus 2^n is faster growing than n^{100} in the limit of very large n.

©2017 UMBC Training Centers

Chapter 2

Asymptotic Notation

Asymptotic Notation

A system of notation for denoting the asymptotic limit of functions.

The notation is called Landau notation, but more commonly is known as Big-Oh notation for the O symbol given to the asymptotic upper-bound of a function.

©2017 UMBC Training Centers

Description

Let $f(n)$, $g(n)$ be real-valued functions, then

$f(n) = O(g(n))$ if and only if $f(n) \leq cg(n)$, for $n \geq n_0$, and $c > 0$.

$f(n) = \Omega(g(n))$ if and only if $f(n) \geq cg(n)$, for $n \geq n_0$, and $c > 0$.

$f(n) = \Theta(g(n))$ if and only if $c_1 g(n) \leq f(n) \leq c_2 g(n)$, for $n \geq n_0$, and $c_1, c_2 > 0$.

Here n_0, c, c_1, c_2 are non-negative constants.

The notation has the following meaning.

$f(n) = O(g(n))$ means $f(n)$ is bounded above by $g(n)$, up to a constant factor, as n increases. This is the upper-bound on the growth of $f(n)$.

$f(n) = \Omega(g(n))$ means $f(n)$ is bounded below by $g(n)$, up to a constant factor, as n increases. This is the lower-bound on the growth of $f(n)$.

$f(n) = \Theta(g(n))$ means $f(n)$ is bounded both above and below by $g(n)$. We say the bounds are "tight" because the upper- and lower-bounds are "close" for sufficiently large n.

Collectively, these are known as “Big-Oh” notation.

This notation is a convenient short-hand for describing the asymptotic limits of a function.

We remark that $O(1)$ refers to constant-time, meaning the value of the function does not change with the input. In describing algorithms it means that the work (runtime) of the algorithm does not change with the input.

©2017 UMBC Training Centers

Asymptotic bounds - Example

Let $f(n) = 2n^3 + 6n^2 + 19n + 1001$.

Then $f(n) = O(n^3)$.

The highest order term dominates the growth of the function as n approaches infinity, thus all lower order terms can be ignored.

In some applications the lower order terms are combined into one notation to denote the next order that follows the highest,

e.g. $f(n) = 2n^3 + O(n^2)$.

Observe that we can write $f(n) = \Theta(n^3)$ because it is both $O(n^3)$ and $\Omega(n^3)$. Specifically, it is bounded above and below by n^3.

But $f(n) = O(n^2)$ is wrong because it leaves out the n^3 term!

Similarly $f(n) = \Omega(n^2)$ is also incorrect.

However, we can say $f(n) = O(n^5)$, but it would not be "tight". Meaning it is correct but not accurate or precise.

©2017 UMBC Training Centers

Little-Oh

We can drop the equality in the definitions for Big-Oh notation, which leads to Little-Oh notation.

$f(n) = o(g(n))$ if and only if $f(n) < cg(n)$, for $n \geq n_0$ and $c > 0$.

$f(n) = \omega(g(n))$ if and only if $f(n) > cg(n)$, for $n \geq n_0$ and $c > 0$.

Thus we call these,

$f(n) = o(g(n))$ "little-oh"

$f(n) = \omega(g(n))$ "little-omega".

These indicate that $f(n)$ approaches but never reaches $cg(n))$.

If $f(n) = n^2$, then $f(n) = o(n^3)$ and likewise $f(n) = \omega(n)$.

But $f(n) = n^3$ is never $o(n^3)$.

Show asymptotic bounds

Show that $n^2 + 2n + 3 = \Theta(n^2)$, for $n \geq 1$.

Specifically, show that $c_1 g(n) \leq n^2 + 2n + 3 \leq c_2 g(n)$, which requires finding c_1, c_2 and $n_0 \geq 1$.

First observe that $n^2 + 2n + 3 \leq n^2 + 2n^2 + 3n^2 \leq 6n^2$ for any positive value of n.

Also observe that $n^2 + 2n + 3 \geq n^2$ as well. Thus,

$$n^2 \leq n^2 + 2n + 3 \leq 6n^2,$$

where $c_1 = 1$ and $c_2 = 6$, $\forall n \geq 1$.

Then by definition $n^2 + 2n + 3 = \Theta(n^2)$.

Notice to get a value for c_2, we had made all terms the same order as the highest term so then we can simply add them and c_2 is just the sum of the coefficients.

This is a simple trick to get Big-Oh notation. Let's apply it to the next example.

©2017 UMBC Training Centers

Show $3n^2 + 5n + 12 = O(n^2)$ for $n \geq 1$.

Let $c = 3 + 5 + 12 = 20$, thus

$$3n^2 + 5n + 12 \leq 3n^2 + 5n^2 + 12n^2 = 20n^2.$$

Therefore $3n^2 + 5n + 12 = O(n^2)$.

That was quite easy. But the point here isn't how we get the constant coefficient.

We can use any method, and in fact, due to the inequality all that matters is that we find some value that satisfies the inequality.

Show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Observe that our earlier trick of making all terms the same order as the highest does not work.

$$\frac{1}{2}n^2 - 3n \not\leq \frac{1}{2}n^2 - 3n^2$$

E.g. Substituting $n = 2$ leads to $-4 \not\leq -10$.

We need to find c_1, c_2, n_0 such that,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

for $n \geq n_0$ an $c_1, c_2 > 0$.

Let's divide out the highest order term.

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Observe $\frac{1}{2} - \frac{3}{n}$ tends to $\frac{1}{2}$ as $n \to \infty$ starting at $n > 0$. Then $\frac{1}{2} - \frac{3}{n} \leq c_2$ for $c_2 = \frac{1}{2}$ and $n \geq 1$.

Now observe $c_1 \leq \frac{1}{2} - \frac{3}{n}$ must hold for $c_1 > 0$. Thus $\frac{1}{2} - \frac{3}{n} > 0$ leads to $\frac{1}{2} > \frac{3}{n}$ so $n > 6$.

©2017 UMBC Training Centers

Choosing $n = 7$ gives $\frac{1}{2} - \frac{3}{7} = \frac{1}{14} > 0$.

Hence $0 < c_1 \leq \frac{1}{2} - \frac{3}{n}$ holds for $c_1 = \frac{1}{14}$, $n_0 = 7$. Therefore,

$$\frac{1}{2}n^2 - 3n = \Theta(n^2) \text{ for } c_1 = \frac{1}{14}, c_2 = \frac{1}{2}, n_0 = 7.$$

One last important note. We are free to choose n_0 and the constants c_1, c_2 to satisfy the inequalities.

Choose $n_0 = 10$:

$$c_1 \leq \frac{1}{2} - \frac{3}{10} \leq c_2 \implies c_1 \leq \frac{1}{5} \leq c_2$$

Then $n_0 = 10, c_1 = \frac{1}{5}, c_2 = 1$, also satisfies the definition for $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Exercise Set 1: Asymptotics

a) Computer scientists use a variety of terminology to describe the time complexity of functions- oftentimes sticking to the ’big oh’ notation. Though Big-Oh can be used for any asymptotic analysis, provide your answers in terms of what the text describes. **Describe Big-O, Big-Θ, and Big-Ω notation. Provide a complete sentence for each, and be sure to note their differences.**

b) As stated above, many computer programmers will generally stick with just Big-Oh notation. You will provide the asymptotic complexity using Big-Oh notation for the following functions, in terms of n.
For example- the function $k(n) = 10n^2$ would be bounded in Big-Oh notation $O(n^2)$.

i. $f(n) = 3n^2 + n + 400$

ii. $g(n) = n + n$

iii. $h(n) = \log_2(10n) + 2n$

iv. $m(n) = (\log_2(2n))^2$

v. $l(n) = 4n^2 + 3n^4 + \log_2(n)$

c) As you will recall, the chapter describes functions that grow ’intractably’.

i. What is the definition of *intractable growth* in terms of algorithms?

ii. What are the three *intractable* functions that the chapter lists?

iii. Provide an example of each of these three- 3 examples total.

d) Describe the circumstances under which two functions are asymptotically equivalent. Provide an example of two functions that are **not the same** but are asymptotically equivalent.

e) Determine the time complexity of the following code snippets, using Big-Oh notation.

i. Determine the time complexity in terms of *n*.

```
int i, j, k;
j = 40;
i += 12;

for (j = 0; j < n ; j++) {
        i += 5;
        i -= n;
        for (k = 4; k < 100; k++) {
                i += 20;
        }
}
```

ii. Determine the time complexity in terms of *n* and *m*.

```
int i, j, k;

for (i = 1; i < n; i *= 2) {
        printf("Just like livin' in  \
          paradise.");
}
for (j = 0; j < m; j++) {
        printf("And I don't wanna go  \
          home!");
        for (k = 0; k < i; k++) {
                printf("No!");
        }
}
```

f) If an algorithm is $O(n^2)$, can it **ever** take cubic time to run to completion? Explain why.

Chapter 3

Abstract Data Type

Abstract Data Type

A mathematical model defined by a set of operations on a collection of data objects.

The abstract data type (ADT) provides an interface for operations on the ADT and its data objects.

The interface specifies what operations can be done.

The implementation is how the operations are done.

The ADT does not depend upon an implementation, hence it is an abstract conceptualization of the logical organization and operations on data.

Separating the specification from implementation gives a consistent definition of how to use the ADT while allowing for many different implementations.

Description

The purpose of an ADT is to facilitate the design and analysis of algorithms and data structures without the burden of gory details of computer languages and architectures.

An ADT is an abstract model for the logical organization and operations on data. It has two primary characteristics that define how to use data represented by the model: i) a data type and ii) operations on the data type.

The definition of a *data type* also applies to an ADT, only ADT is an abstract representation for a collection of objects and operations.

Here behavior refers to the result of how the ADT is used, meaning what happens when the operations defined by the ADT are applied.

One can think of an ADT as a logical container that encapsulates both primitive data types and a set of operations on them.

Here are some preliminary definitions that we will use going forward.

data type

A data type is defined by the values it can take and the operations on it.

object

A group of fields or attributes for holding data types and connecting with other objects; also known as a cell.

data structure

An organization of objects for storing data and supporting specific operations on the data structure and its data types.

Integer ADT

Let's consider a simple example of an ADT.

The set of integers given by $\mathbb{Z} = -\infty, \ldots, -1, 0, 1, \ldots, +\infty$ is an ADT. It supports the operations of addition, subtraction, multiplication, and division.

The set of integers is closed under these operations with the exception of division.

> **Note**
>
> A set is *closed* under an operation if that operation on any member of the set results in a member of the set.
>
> Dividing 1 by 2 results in the rational number $\frac{1}{2} = 0.5$, which is not a member of $\mathbb{Z}$.

Integers obey the rules of associativity, commutativity, and distributivity.

Integers are also comparable under the relational operators for i) less than ($<$), ii) greater than ($>$), and iii) equality ($=$).

Together, the values of the integers and the operations on the integers define an ADT.

The integer ADT does not dictate how integers are stored in a computer or how the operations are implemented.

©2017 UMBC Training Centers

For example, an integer can be represented in base-2 or binary representation, and within this it can be one’s or two’s complement.

There are also different methods on how to perform the arithmetic and logical operators.

But the use, and corresponding expectation of results, on integers is the same regardless of how it is managed in a computer.

Abstraction

The ADT provides an interface that defines how it can be used, specifically for invoking the operations of the ADT.

Observe that the interface is only a specification, it does not dictate how the ADT's underlying data structure or its operations are implemented.

An ADT does not define a data structure.

Thus the interface, and subsequently the behavior, of the ADT is the same regardless of the implementation.

Consider a set ADT. The set is an unordered collection of data types. It can hold more than one kind of data type. In addition to standard set-theoretic operations, it also allows operations for adding and removing members, but does not allow duplicates.

This set ADT does not define a data structure. Only a logical organization of the data, namely that the data is unordered, can be of multiple types, and does not have duplicates.

Note

Data structures for a set can be any of the following.

- binary search tree
- hash table
- trie

Custom ADT

There are no hard rules for defining a new ADT.

What matters is deciding the desired outcome of operations on some kind of data.

Suppose we wish to design an ADT that holds a set of socks for wearing on our feet, as opposed to on our head.

The ADT should permit us to add and take out socks, and importantly that it returns socks that are always paired correctly — assuming we don't have two left feet!

The data type here is a sock and the values it can take are a composite of {color, material, pattern}.

A sock can be {blue, cotton, polka dots}.

Hence we want the operations to work on the sock data type. The interface for these operations allows us to use the sock ADT.

The interface is then:

put<t>(x) Add a pair of socks, x, where x has value t.
get<t>() Return a pair of socks that match set t.
count<t>() Return the count of sock pairs that match set t.

©2017 UMBC Training Centers

This interface remains the same whether the ADT is implemented using a mechanical machine that looks like a sock drawer with a robotic arm, or on a computer.

A library developer is then free to choose the algorithms and implementations that carry out the operations.

How would you define an ADT for a piggy bank?

Operations can include *insert*, *shake*, and finally *break*.

This page intentionally left blank.

CHAPTER 4

Array Abstract Data Type

Array

An **abstract data type (ADT)** for a fixed-size sequence of data elements.

Data is logically organized in a linear sequence of cells; a one-dimensional array.

Thus the cells are sequentially ordered.

Each element in the array has a position or index that denotes its order and therefore its cell index as an integer number.

Hence data is organized as a finite sequence of data elements stored in cells.

Then the array has finite bounds with start and end cells.

For n data elements, the ordinals are numbered from 1 to n or 0 to $n - 1$ if using zero-indexing.

Thus each data element can be accessed by its integer cell index, or its offset from the beginning of the array.

All data elements in an array are of the same data type.

©2017 UMBC Training Centers

Description

An array is one of the simplest ADTs for data.

The ADT is defined by a basic set of operations on a fixed-size sequence of cells.

The data elements are logically organized by position in a one-dimensional array (or matrix) of contiguous cells that accepts data of one type, e.g. an array of integers.

Thus any data element can be accessed by its cell position.

The basic operations on the array are to get and set the value of a cell.

Since the array is fixed-size then new cells cannot be added or removed.

This ADT closely resembles the conventional array in many programming languages that is often implemented as an array of contiguous memory cells.

Interface

The interface for an array is minimal. A user accesses any data element in the array by referencing the cell index (ordinal subscript) or an offset counted from the start of the array.

For example, if A is the name of the array then $A[3]$ can refer to the

©2017 UMBC Training Centers

third element (or fourth if zero-indexing).

Note that there is nothing special about the $[\cdot]$ operator, it is simply an interface. The array ADT could use instead the function *get(i)* to access the element in the i^{th} cell.

Both the data organization and operations allowed by the interface are abstract. We only know that the cells are contiguous and we can access an element by the array/cell index.

It is possible that a computer implementation of the array ADT does not place the cells in contiguous memory. Accessing a data element could require a traversal from the beginning of the computer memory that holds the start of the array, rather than simple pointer arithmetic if the cells were laid in contiguous memory addresses, e.g. $^*(A + 3)$.

We will use array index and cell index interchangeably going forward.

Example array ADT

Suppose we have an array ADT for integers using zero-indexing. Let A be the name or handle to this array. Then the value of the third element in the array is obtained by using the array index reference, $A[2]$.

We can iterate through all elements of the array by simply incrementing a counter for each array index. Let i be a counter initialized to zero, then we can access the first five elements using the following algorithm.

1. initialize i to zero and n to five: $i \leftarrow 0$, $n \leftarrow 5$.
2. print $A[i]$
3. increment i by one; set i := i + 1
4. if $i < n$ then go to step 2, else halt.

Figure 4.1 illustrates the array ADT for integers where the cells hold the values and the numbers under each cell is the array index.

$$\begin{aligned} A[0] &= 6 \\ A[1] &= 4 \\ A[2] &= 3 \\ A[3] &= 4 \\ A[4] &= 4 \\ A[5] &= 3 \end{aligned}$$

$$A = \begin{array}{|c|c|c|c|c|c|} \hline 6 & 4 & 3 & 4 & 4 & 3 \\ \hline \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1} & \multicolumn{1}{c}{2} & \multicolumn{1}{c}{3} & \multicolumn{1}{c}{4} & \multicolumn{1}{c}{5} \end{array}$$

Figure 4.1: Array ADT for integers.

Dynamic array ADT

The array ADT described earlier is for managing a fixed-size sequence of data elements.

It was "static" in the sense that the number of cells was fixed and could not be changed without destroying the array and creating a new one with a different size.

Yet, there is no such conceptual limitation. Since it is an abstract model, all that is needed is an operation to grow the array as needed.

Recall that an ADT is primarily a set of operations on the model and the data it represents.

The interface to the ADT specifies how to use the ADT, meaning the interface specifies which operations are available for that ADT.

Let us define a Dynamic Array ADT.

Interface

The dynamic array ADT is similar to the basic array ADT in that the data is logically organized in a one-dimensional array of contiguous cells of homogeneous type.

Let the following interface specify the operations on the ADT.

create(n) Create a dynamic array of size $2n$ to hold n elements and an extra n capacity for the next n elements.

destroy() Destroy the dynamic array.

get(i) Get the value at index i.

set(i,x) Set the value at index i to x.

length() Get the length (number of elements) of the dynamic array.

grow() Increase the capacity by another n cells.

add(x) Add value x to the end of the array.

remove() Remove the value at the end of the array.

This dynamic array will automatically grow as needed. The interface given here is for instruction and is only one of many possible.

Example dynamic array ADT

Suppose now we have the sequence of five integers that we wish to add to the dynamic array. Using the interface, we create the dynamic array for five elements and then we add each integer.

Assume the array begins with index zero, we can print out all values in the dynamic array using the following print algorithm.

```
Require: A                                        ▷ dynamic array
function PRINTALL
    for $i = 0$ up to A.length() do
        print A.get(i)
```

Now we want to add ten more elements. The initial capacity of the array is ten, but the first five cells are occupied. Since this ADT will automatically invoke its *grow()* function to increase the capacity by another five elements, we as the user can freely add the elements.

```
Require: A                                    ▷ dynamic array
    initialize n to ten; $n \leftarrow 10$
    function INPUT
        query user to enter an integer
        return integer given by user
    for $i = 0$ up to $n$ do
        set x := INPUT
        A.add(x)
```

Observe that we have not described how the dynamic array adds more capacity or specified the data structure. Those are implementation details.

Chapter 5

List Abstract Data Type

List An **abstract data type (ADT)** for a sequence of data elements.

Data is logically organized in a linear sequence of cells.

Thus the cells are sequentially ordered.

The sequential ordering of cells leads to the following properties.

- Each element can be located by the integer ordinal that denotes its position in the list.
- From any element in the list, the next element can be located.

The list ADT supports the operations for locating, adding, and removing elements from any position in the list.

Thus the list can grow and shrink as needed.

Description

The list ADT is defined by a set of operations on an ordered sequence of cells.

The cells are ordered sequentially, thus any data element can be found by its position in the list and each cell is connected to the next in the sequence.

This allows the list ADT to be used for queries such as, “get the fifth element”.

Operations on the list include locate, add, and remove any element. Hence the list can grow and shrink automatically.

These operations leave the list intact.

The list ADT can also include operations to split and concatenate lists.

A common data structure is a linear network of interlinked-nodes, better known as a linked list.

We will refer to the cells as nodes since that is the common term for many of the data structures used to implement lists.

Interface

The list ADT interface can include the following.

©2017 UMBC Training Centers

create() Create an empty list.
destroy() Destroy the list.
head() Return the start of the list.
add(x,p) Add value x at position p.
remove(x,p) Remove value x at position p.
get(p) Return the value at position p.
set(p,x) Set the value at position p to x.
position(x) Return the first position of value x.
next() Get the next position in relation to the current position.
empty() Returns a Boolean to denote if the list is empty.

This is an example interface of common functions that is meant only as a representative of possible operations.

The interface for the list ADT specifies how to use the list, specifically which operations are permitted on the ADT.

Example list ADT

Consider a sequence of integers that we want to represent by the list ADT. We will assume the positions begin at zero. Let L be the name for this list.

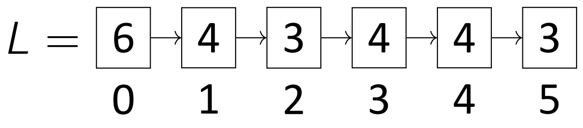
We can iterate through all elements of the list by starting at the head and requesting the next element until we reach the tail.

> We could also use a counter i for each position and increment to get the next in the list.

Here is a basic algorithm on the list ADT to print all values in the list.

```
Require: L                                  ▷ list ADT
set p := 0
while L.get(p) is not null do
    print L.get(p)
    set p := L.next()
```

Figure 5.1 illustrates the list ADT for integers where the cells hold the values and the numbers under each cell is the position index.



$$L.\text{get}(0) = 6$$
$$L.\text{get}(1) = 4$$
$$L.\text{get}(2) = 3$$
$$L.\text{get}(3) = 4$$
$$L.\text{get}(4) = 4$$
$$L.\text{get}(5) = 3$$

Figure 5.1: List ADT for integers.

©2017 UMBC Training Centers

Fisher-Yates Shuffle

Fisher-Yates Shuffle

An algorithm for randomly permuting a finite sequence of numbers.

The permutation is unbiased meaning any permutation is equally likely.

Given a sequence of numbers, the algorithm randomly removes a number from the sequence and repeats until no numbers remain.

The resulting sequence of removed numbers is an unbiased permutation.

Named after its inventors, Ronald Fisher and Frank Yates.

Invented in 1938, the Fisher-Yates Shuffle was independently re-discovered by Richard Durstenfeld in 1964 and later became known as the Knuth Shuffle.

Description

Consider a sequence of n numbers that we wish to randomly shuffle.

Then counting from zero, each number is associated with an index from $0..n-1$ denoting its position in the list.

The Fisher-Yates algorithm iterates backwards through the list keeping a descending counter i for each iteration step. At each step i it selects a random position index k such that $0 \leq k \leq i$. It then exchanges the numbers at k, i.

The Fisher-Yates algorithm using our list ADT is as follows.

```
Require: L                                              ▷ list ADT
for $i = n - 1$ to $0$ do
    set k := random number such that $0 \leq k \leq i$.
    set j := L.get(i)
    L.set(i,L.get(k))
    L.set(k,j)
```

It's easy to see that the algorithm should take $O(n)$ time. But a naïve implementation that walks from the head of the list to each position every step will lead to $O(n^2)$ time.

The algorithm pseudocode uses our list ADT interface for the *get* operation, but does not rely on how it works.

It is conceivable that *get(p)* can take $O(1)$ time given a suitable data structure.

> The list can be implemented using an array data structure that provides random access to each element in constant-time.

Exercise Set 2: Abstract Data Types

a) Abstract data types are immensely useful to programmers, especially when designing products at scale. From a theoretical standpoint, explain **why** we omit the implementations when we design an abstract data type.

b) You have learned the basics of an abstract data type thus far. Now, you are expected to design an abstract data type for each of the following. That is, provide at least **5 operations** (referred to as an *interface* in the chapter) and **3 variables/properties** that each of these abstract data types have. Additionally, provide at least **2 implementations**.

For example, if you are asked to describe a **sports game**, an operation could be `get list of players`, a variable/property could be `current score`, and an implementation could be a `soccer game`.

i. Bank Account

ii. Student Roster

iii. Motor Vehicle

c) Provide 3 examples of abstract data types specific to computer science, listed in the textbook (or otherwise, if you prefer).

©2017 UMBC Training Centers

CHAPTER 6

Linked List

Linked List

A **data structure** for implementing the list abstract data type (ADT).

A linked list is a linear sequence of connected cells, where each cell has a pointer to the next.

A cell is located by traversing to its position in the list, hence the linked list is a sequential access data structure.

The linked list supports the operations specified by the list ADT, including locating, adding, and removing objects from any position in the list.

Thus the linked list can grow and shrink as needed.

©2017 UMBC Training Centers

Description

The linked list is a fundamental data structure for the linear storage of data.

The data structure is simply a sequence of objects linked together, i.e. a chain of objects.

At a basic level, each object holds a value for some data type, and a pointer to the next object in the list.

> This is a *singly linked list*.
>
> Adding a pointer to the previous object makes it a *doubly linked list*.

The entire list can be traversed from the start by following the next object in the sequence until there isn't a next object.

The position of any object in the list is known simply by counting from the start of the list.

Any element can be removed or added to the linked list making it a dynamic and flexible data structure.

This alone makes it a common choice for implementing many different ADTs.

©2017 UMBC Training Centers

Note

The following is a sample of ADTs that are supported by singly and doubly linked list data structure.

- bag
- dynamic array
- stack
- queue

A linked list resembles a sequential network and so the objects are often to referred to as nodes.

We will use object/cell/node interchangeably for the data elements in linked lists.

Design

Each cell in a linked list has the following.

- data element
- pointer to the next cell
- pointer to the previous cell (if doubly-linked)

A linked list takes values according to the data type that it encapsulates, but only for a single data type. Hence it is a collection of homogeneous type.

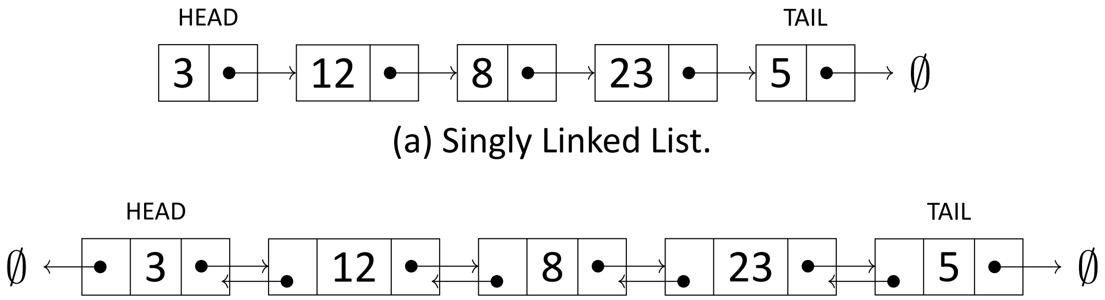
The start and end of the list are known respectively as the *head* and *tail*.

The tail is the last element in the list so its next pointer points to null.

©2017 UMBC Training Centers

Some instances have a special *header* node that contains no data but a pointer to the start of the list. But in many applications the header is extraneous.

Figure 6.1 depicts both singly and doubly linked lists holding integers.



(a) Singly Linked List.

(b) Doubly Linked List.

Figure 6.1: Linked List.

©2017 UMBC Training Centers

Singly vs Doubly Linked List

Each node in a doubly linked list has a pointer to its previous and next neighbors.

This allows traversal in both directions and is the primary benefit in comparison to the singly linked list.

But for a list of n nodes, the added pointer to the previous node requires $O(n)$ more space than a singly linked list.

The doubly linked list also has the distinct advantage that if given a node in the list, it can remove it in constant-time since it can re-link the previous and next neighbors of the removed node.

The following is a summary of the primary benefits of the doubly linked list.

- Bi-directional traversal.
- Deletion of a given node in $O(1)$ time.

Both the singly and doubly linked list can be used to implement the list ADT interface.

The common operations between the two linked lists have the same asymptotic bounds.

The following table lists a comparison of the time complexity for operations between a doubly linked list (DLL) and singly linked list (SLL).

	insert/delete front	insert/delete end	insert/delete middle	delete node	find
DLL	$O(1)$	$O(n)$, $O(1)$ at tail	$O(n)$, $O(1)$ at pos.	$O(1)$	$O(n)$
SLL	$O(1)$	$O(n)$, $O(1)$ at tail	$O(n)$, $O(1)$ at pos.	N/A	$O(n)$

Figure 6.2: Time complexity comparison.

©2017 UMBC Training Centers

Linked List vs Array

Both the linked list and array data structures store data as a linearly ordered sequence, and both can support the list ADT.

An array data structure has the following advantages over a linked list.

- Random access to any cell.
- Cells are stored contiguously in memory.
- More compact storage.

These can lead to improved performance in practice because of better memory cache effects.

In contrast, each node in a linked list must hold not only a data value but at least one pointer to the next node in the list.

> **Note**
>
> A linked list storing 8 byte integers on a 64-bit machine requires $2\times$ more space than that of an array, and if doubly linked then it takes $3\times$ more space.
>
> A pointer is 8 bytes on a 64-bit machine.

The nodes in a linked list are sequentially accessed. A pointer from one node to an adjacent node provides the connectivity.

But this makes the linked list a more flexible data structure with the following advantages.

- Not fixed-size; can be as large as needed.
- Grow and shrink on-demand.
- Concatenation with other lists.

An array cannot be concatenated, instead the contents must be copied to a new array.

The following is a short summary comparison of the respective advantages of arrays and linked lists.

Array advantages

- Random access to any cell.
- Cells are stored contiguously in memory.
- More compact storage.

Linked list advantages

- Not fixed-size; can be as large as needed.
- Grow and shrink on-demand.
- Concatenation with other lists.

Runtime comparison

Consider a sequence of n data elements stored in either an array or linked list.

Adding an element in the middle of an array requires copying and shifting all the affected elements to the end, but only if the array had extra space. This takes $O(n)$ time.

Removing an element from the middle of an array also requires copying and shifting affected elements but not extra space, instead the space at the end is left unused. This takes $O(n)$ time.

> **Note**
>
> If order does not need to be preserved, then array deletion can be done in $O(1)$ time!
>
> ---
>
> Swap the deleted element with the last in the list and decrement the size count.

In contrast, if we are already at some position, then it takes $O(1)$ time to insert or delete a cell at the next position.

The following table lists a comparison of the time complexity for common operations between an array and linked list. Assume that insertion and deletion at the beginning and middle for an array also entail additional complexity- namely, in terms of the space required for insertion.

	insert/delete front	insert/delete at end	insert/delete middle	find	index
array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$
linked list	$O(1)$	$O(n)$, $O(1)$ at tail	$O(n)$, $O(1)$ at position	$O(n)$	$O(n)$

Figure 6.3: Time complexity comparison.

Chapter 7

Linked List Operations

The linked list data structure supports the operations of the list ADT interface.

The data structure is an explicit instance of the logical ordering of the list ADT.

The linked list is a manifest instance of an abstract model and the operations on that model.

The linked list stores data as a sequence of *n* cells with the following properties.

- Each cell can be located by its position in the list.
- From any cell in the list, the next cell can be located.

Primarily, the linked list supports the operations to locate, add, and remove data nodes from any position in the list.

List Interface

The following is a representative list interface that is commonly implemented using the linked list data structure.

create(x)	Create a list node with value *x*.
destroy()	Destroy the list.
add_front(x)	Add value *x* to the front.
add_back(x)	Add value *x* to the back.
add_at(x,p)	Add value *x* at position *p*.
remove_front()	Remove front node.
remove_back()	Remove back node.
remove_at(x,p)	Remove value *x* at position *p*.
size()	Get the size of the list.
print()	Print each value stored in the list.

The list interface specifies what operations can be done on a list, but does not describe how the operations are done.

> The ADT interface states what can be done but not how to do it.

The linked list data structure provides the *how* part.

Illustration of Operations

We'll illustrate the implementation of some list operations on the linked list data structure.

Let *n* denote the size of the linked list, specifically the number of nodes chained together.

Recall that each node in the linked list has a pointer to the next node. All of the operations can be implemented by following pointers and changing pointer targets.

We'll use a $\emptyset$ in a node to indicate a null pointer, rather than explicitly pointing to a free-floating null symbol.

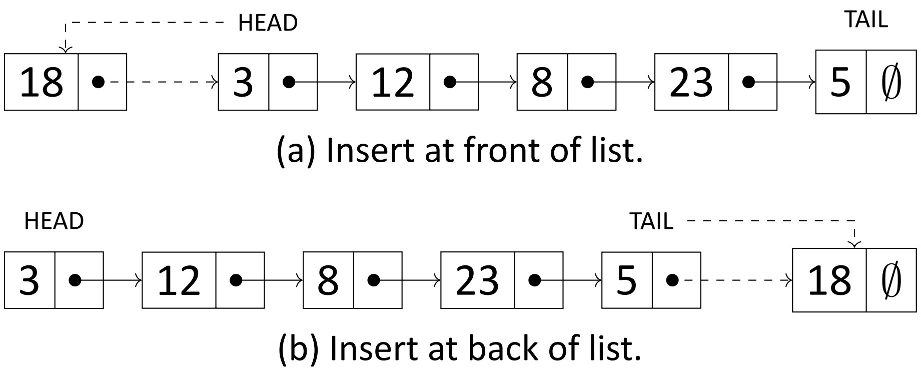
Add front/back

Let's begin with adding a new node to the front or back of the list.

Adding to the front is a simple task of setting the new node's next target to the head of the list. This takes $O(1)$ time.

Adding to the back requires traversal to the tail, which takes $O(n)$ time. Once at the tail it takes $O(1)$ time to set the last node's next target to the new tail node.

The following figure depicts these operations.



(a) Insert at front of list.

(b) Insert at back of list.

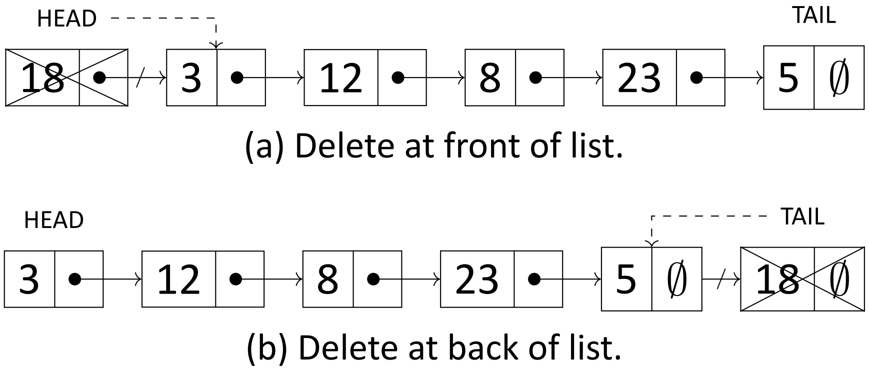
Remove front/back

Removing from the front and back of the list is similar to adding.

Removing the head node is just a matter of destroying that object. Any pointer handle to the list is updated to the new head.

Removing the tail requires traversal up to the last node and setting the target of the $n-1$ node's next pointer to null.

The following figure depicts these operations.



(a) Delete at front of list.

(b) Delete at back of list.

Add/remove middle

Adding anywhere in between the front and tail nodes requires traversal up to the position prior to the node being added or removed. This takes $O(n)$ time.

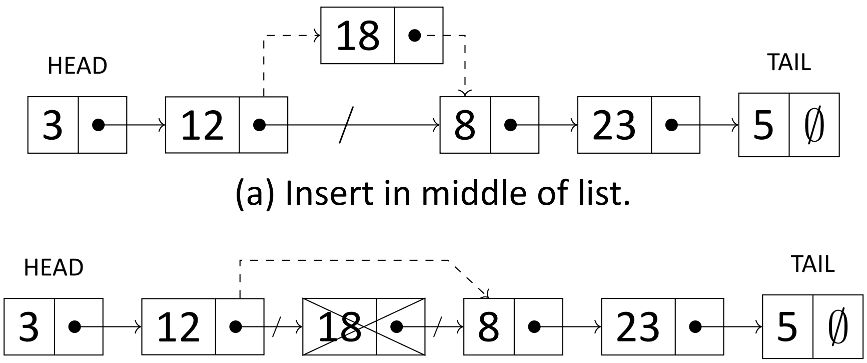
If the node to be added or removed is at position p, then we traverse to position $p - 1$.

On insertion, the next pointer of node $p - 1$ points to the inserted node, and the next pointer of the inserted node points to the $p - 1$ node's old next target.

On deletion, the $p - 1$ node's next pointer points to the target of the deleted node's next pointer. The deleted node is then destroyed.

The previous pointers of affected nodes must also be updated if a doubly linked list.

The following figure depicts these operations.



(a) Insert in middle of list.

(b) Delete in middle of list.

©2017 UMBC Training Centers

Chapter 8

Linked List Implementation

The linked list data structure is a sequence of objects that are chained together using pointers to point from one node to the next.

Traversing the linked list is performed by successively moving from one node to the target of that node's next (or previous) pointer.

Changing the target of a node's pointer is a constant-time action. The target can either be set to a new node or to null if the node is an end-point of the list.

Implementing the list ADT interface using a linked list data structure is therefore accomplished by pointer traversal and pointer updating.

Interface

We will discuss potential algorithms for the following representative list interface.

create(x)	Create a list node with value x.
destroy()	Destroy the list.
add_front(x)	Add value x to the front.
add_back(x)	Add value x to the back.
add_at(x,p)	Add value x at position p.
remove_front()	Remove front node.
remove_back()	Remove back node.
remove_at(x,p)	Remove value x at position p.
size()	Get the size of the list.
print()	Print each value stored in the list.

Recall that an ADT interface is only a specification. The implementation is separate.

This separation of interface from implementation ensures that the expected behavior and use of the ADT remain consistent. This allows flexibility in how the interface operations are implemented.

Thus an application that uses the interface can change the implementation without breaking interoperability with other applications that depend upon it.

In the next passages we'll give algorithms for the interface specified here, but the reader should note that they can replace these algorithms with their own.

Data type

Let's begin first with defining our linked list data structure type. This compound type is the data type of every node in the linked list.

Let LIST be the name of our linked list data type. Let t be a primitive type, e.g. integer, of the LIST data member, meaning each LIST node encapsulates data of type t.

Thus the values that LIST stores depend on the data type t. Given any node in LIST, the operations allowed on t can be applied to the data member of that node.

For simplicity, we will consider only the singly linked list.

Our LIST data type has the following members:

- data: a data element of type t
- next: a pointer to type LIST

©2017 UMBC Training Centers

Algorithms

Next we'll give the algorithms that show how to implement a subset of the list interface described earlier.

Observe that the linked list data structure is recursively defined.

Any sublist in a linked-list is itself a linked list, and a sublist can be a single node.

We will give recursive algorithms going forward.

Note that although we give pseudocode, it can be easily translated into a working implementation.

We leave out details that are specific to any programming language. But observe that each node in LIST is implemented as a composite type such as a *struct* or *class*, so the reader can invoke their favorite programming language for managing each node.

It should be clear that once a LIST object is created, it requires a new set of resources from the computer. Deleting a LIST object releases the resources. We will use the terms *new* and *delete* to refer respectively to allocating and deallocating these resources.

The first operation is of course creating a LIST node. But this is not an algorithm, rather it is a request for allocating resources for type LIST.

Thus the implementation of the *create(x)* interface performs the following:

1. set data := new x
2. set next := $\emptyset$

We will use $\emptyset$ to denote a null pointer.

Mutable functions

Add to front of the list.

```
function ADD_FRONT(LIST head, LIST x)
    set x$\rightarrow$ next := head;
    set head := x
```

Add to back of the list.

```
function ADD_BACK(LIST node, LIST x)
    if node$\rightarrow$next equals $\emptyset$ then
        set node$\rightarrow$next := x
        return
    return ADD_BACK(node$\rightarrow$next, x)
```

Destroy the list.

```
function DESTROY(LIST node)
    if node equals $\emptyset$ then return
    set next := node$\rightarrow$next
    delete node
    set node := next
    return DESTROY(node)
```

Immutable functions

Get the size of the list.

```
function SIZE(LIST node)
    set s := 1
    if node$\rightarrow$next equals $\emptyset$ then return s
    return s := s + SIZE(node$\rightarrow$next)
```

Exercise Set 3: Linked Lists

a) Suppose you have a very important job, and you have chosen a person, Person A, to perform that role. The job is, in fact, so important, that you asked Person A to select a backup. You have also requested that the 'backup' for Person A select a backup. You enforce this up until the point where the original person selected, Person A, has 49 backups. Today, Person A has provided you with a list detailing their backup, their backup's backup, their backup's backup's backup, etc.

Being the computer scientist that you are, you have decided to represent this chain of backups as a linkedlist of Strings (stored as character arrays). Person A is represented by the first element, and they maintain a pointer to their 'backup', who maintains a pointer to their 'backup', and so on and so forth.

You want to quickly validate that this chain of backups is valid. That is, you want to find out if anybody in the chain of backups has referenced someone who has already signed up to be a backup. For example. If Person D is backup for Person C who is backup for Person B, who is ultimately backup for person A, then Person D's backup **cannot** be Person A, B, C, or D.

Provide a C function `backup_validator(Node *)` that takes in a linked list of 'backups' stored as character arrays, and checks whether or not the list of backups provided is indeed valid. In order to receive full credit, solve with problem *without any additional space complexity*.

©2017 UMBC Training Centers

b) Implement a singly-linked integer list with the following functions. You are welcome to create your own additional subroutines to help.

  - Add (to end)

  - Remove (all occurrences of a given element)

  - Search

c) Implement a doubly-linked **sorted** linked integer list with the following functions. Recall that a sorted linkedlist maintains sorted order of all elements within it. You will be maintaining ascending sorted order. You are welcome to create your own additional subroutines to help.

  - Add (maintaining sorted order)

  - Remove (all occurrences of a given element)

  - Search

Chapter 9

Stack

Stack

An **abstract data type (ADT)** for a sequence of data elements that are added and removed in Last-In-First-Out (LIFO) order.

Data is logically organized in a linear sequence of cells with a conceptual *top* wherefrom data elements are added and removed.

Thus a stack is either empty or it is a stack with a top.

The stack ADT has two primary operations, i) add a new element to the top of the stack, ii) remove an element from the top of the stack.

The stack grows and shrinks as needed.

Description

The stack ADT is defined by a limited set of operations on an ordered sequence of cells.

A cell is a container or object that holds a value of some data type. We will use cell/object/node interchangeably.

The principal significance of the sequential ordering of cells is that the first item to be removed was the last one added — Last-in-First-Out (LIFO).

Operations on the stack are for primarily adding and removing data but only from the top of the stack, commonly called *push* and *pop*, respectively.

Conceptually a stack ADT is like a stack of pancakes.

> New pancakes are piled on top of the stack, wherefrom the diner removes pancakes for eating.

Figure 9.1 depicts the logical organization and fundamental operations of a stack.

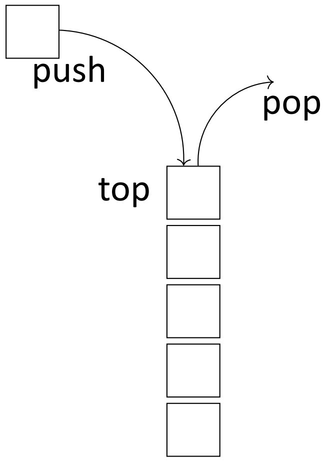


Figure 9.1: Stack.

CHAPTER 10

Stack Interface

The stack ADT interface specifies the two primary operations for adding and removing items from the stack in LIFO order.

These operations are commonly called *push* and *pop* for adding and removing items, respectively.

Another common operation permits a user to inspect the topmost cell without modifying the stack. The interface for this is commonly known as *peek* or *top*.

These operations make up the common interface of the stack ADT, given next.

create()	Create an empty stack.
destroy()	Destroy the stack.
top()	Return the top of the stack but do not remove it.
push(x)	Add value x to the top.
pop()	Remove the topmost value.
size()	Return the number of items in the stack.
empty()	Returns a Boolean to denote if the stack is empty.

The interface for the stack ADT specifies how to use the stack, specifically which operations are permitted on the ADT.

Operations

The operations of the stack ADT primarily interact with the top of the stack.

A suitable data structure should take $O(1)$ time for adding, removing, and inspecting an item at the top.

The data in a stack is logically organized as a linear sequence of cells.

Thus each item added to a stack has a position in the stack corresponding to the size of the stack (height) at the time the item was pushed onto it.

> The size of the stack is the difference between the number of *push* and *pop* operations starting from an empty stack.

The LIFO order of the stack makes it useful for many operations in which the most recent item should be accessed first.

> **Note**
>
> Applications of the stack ADT:
>
> - computer program function call stack
> - reversing a sequence (push all first, then pop until empty)
> - backtracking algorithms, e.g. Depth-First Search
> - calculators

Figure 10.1 illustrates a sequence of push and pop operations on a stack.

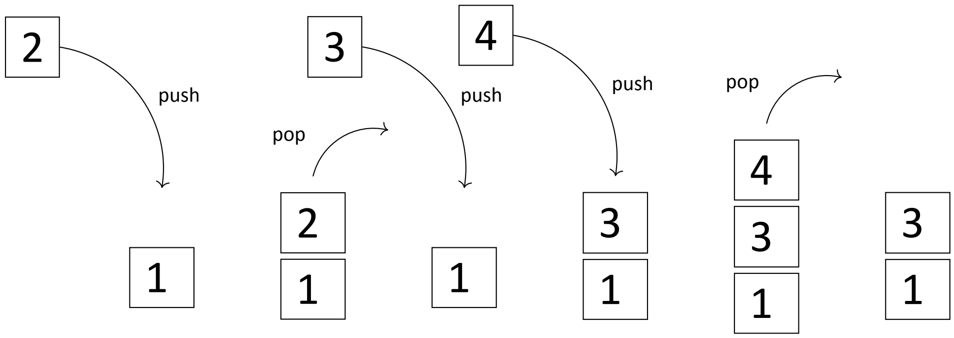


Figure 10.1: Sequence of stack operations.

©2017 UMBC Training Centers

Stack ADT - Postfix calculator

The stack ADT is often used for postfix calculator operations.

We typically write arithmetic expressions in infix notation where each binary operator is between the two operands, e.g. $2 + 3$.

In postfix notation the operands precede the operator, e.g. $(2\,3\,+)$.

> In prefix notation the operator precedes the operands, e.g. $(+\,2\,3)$.

It more efficient for a computer to process expressions in postfix notation because operators appear in order of their precedence.

A stack is well-suited for calculations in postfix notation.

We will use the stack interface to first convert from infix to postfix notation and then again to evaluate the postfix expression.

Infix-to-postfix conversion

The algorithm for infix-to-postfix conversion using the stack ADT interface is given next. The output is a new postfix expression.

©2017 UMBC Training Centers

```
Require: A stack.
    Input: infix expression
    Output: postfix expression
    Operator precedence in increasing order of priority: $(,), +, -, *, /, \hat{}$
    Items removed by pop are written to the output except for “(”.
    for all symbols $S$, reading from left-to-right until completion do
        if $S$ is an operand then write to output.
        if $S$ is an operator then
            if $S$ is higher priority than the operator returned from top then push $S$ onto stack.
            Otherwise pop until reaching a lower priority symbol then push $S$ onto stack.
        if $S$ is “(” then push $S$ onto stack.
        if $S$ is “)” then pop all items up to and including “(”.
    On completion, pop each item from the stack.
```

Note

The priority of parentheses for the infix to postfix conversion is not related to the conventional operator precedence rules (e.g. PEMDAS). They are used for catching groupings of operands and operators in the conversion, and then discarded since they are not needed in postfix expression evaluation.

Infix-to-postifx example

Let's apply the conversion on the following infix arithmetic expression: $6 + 4(2 * 3 - 1)/5$.

Don't forget the implicit multiplication operator between an operand and "("!

Table 10.2 demonstrates the conversion.

step	read	stack operation	write	expression
1)	6		6	6
2)	+	push(+)		6
3)	4		4	6 4
4)	∗	push(∗)		6 4
5)	(	push("(")		6 4
6)	2		2	6 4 2
7)	∗	push(∗)		6 4 2
8)	3		3	6 4 2 3
9)	−	pop(∗), push(−)	∗	6 4 2 3 ∗
10)	1		1	6 4 2 3 ∗ 1
11)	)	pop(−), pop("("), discard parens	-	6 4 3 2 ∗ 1 −
12)	/	push(/)		6 4 3 2 ∗ 1 −
13)	5		5	6 4 3 2 ∗ 1 − 5
14)		pop(/), pop(∗), pop(+)	/ ∗ +	6 4 2 3 ∗ 1 − 5 / ∗ +

Figure 10.2: Infix conversion to postfix.

Postfix evaluation

Given an arithmetic expression in postfix notation, we can use the stack interface to evaluate the expression.

The algorithm for postfix evaluation using the stack ADT interface is given next.

Previously we had converted an example arithmetic expression from

```
Require: A stack.
  Input: postfix expression
  Output: result
  for all symbols $S$, reading from left-to-right until completion do
    if $S$ is an operand then push $S$ onto stack.
    if $S$ is an operator then
      set $y$ := item from pop
      set $x$ := item from pop
      push result of $xSy$ onto stack
  On completion, pop last item and output.
```

infix to prefix notation:

infix: $6 + 4(2 * 3 - 1)/5$

postfix: $6\,4\,2\,3\;*\;1\;-\;5\,/\;*\;+$

The above expression should evaluate to 10.

Table 10.3 demonstrates the evaluation.

step	read	stack operation	stack (grows right)
1)	6	push(6)	6
2)	4	push(4)	6 4
3)	2	push(2)	6 4 2
4)	3	push(3)	6 4 2 3
5)	$*$	y=pop(3), x=pop(2), push($x * y = 6$)	6 4 6
6)	1	push(1)	6 4 6 1
7)	$-$	y=pop(1), x=pop(6), push($x - y = 5$)	6 4 5
8)	5	push(5)	6 4 5 5
9)	$/$	y=pop(5), x=pop(5), push($x/y = 1$)	6 4 1
10)	$*$	y=pop(1), x=pop(4), push($x * y = 4$)	6 4
11)	$+$	y=pop(4), x=pop(6), push($x + y = 10$)	10

Figure 10.3: Evaluate postfix expression.

CHAPTER 11

Queue

Queue

An **abstract data type (ADT)** for a sequence of data elements that are added and removed in First-In-First-Out (FIFO) order.

Data is logically organized in a linear sequence of cells with a conceptual *front* and *back* wherefrom data elements are removed and added, respectively.

An empty queue is still a queue.

The queue ADT has two primary operations, i) add a new element to the back of the queue, ii) remove an element from the front of the queue.

The queue grows and shrinks as needed.

Description

The queue ADT is defined by a limited set of operations on an ordered sequence of cells.

A cell is a container or object that holds a value of some data type. We will use cell/object/node interchangeably.

The principal significance of the sequential ordering of cells is that the first item to be removed was the first one added — First-in-First-Out (FIFO).

Operations on the queue are for primarily adding and removing data but only from the ends of the queue, commonly called *enqueue* and *dequeue*, respectively.

Conceptually a queue ADT behaves like its namesake.

> Individuals get in line (queue) at a first come, first serve counter.

Figure 11.1 depicts the logical organization and fundamental operations of a queue.

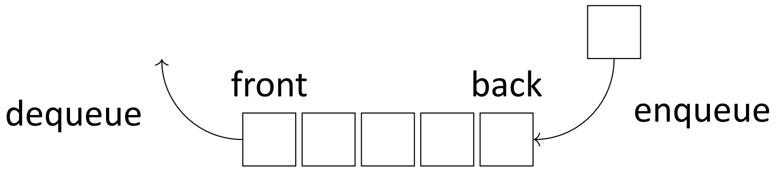


Figure 11.1: Queue.

This page intentionally left blank.

Chapter 12

Queue Interface

The queue ADT interface specifies the two primary operations for adding and removing items from the queue in FIFO order.

These operations are commonly called *enqueue* and *dequeue* for adding and removing items, respectively.

Another common operation permits a user to inspect the firstmost cell without modifying the queue. The interface for this is commonly known as *peek* or *front*.

These operations make up the common interface of the queue ADT, given next.

create() Create an empty queue.
destroy() Destroy the queue.
front() Return the front of the queue but do not remove it.
enqueue(x) Add value x to the top.
dequeue() Remove the firstmost value.
size() Return the number of items in the queue.
empty() Returns a Boolean to denote if the queue is empty.

The interface for the queue ADT specifies how to use the queue, specifically which operations are permitted on the ADT.

Operations

The operations of the queue ADT primarily interact with the ends of the queue.

A suitable data structure should take $O(1)$ time for adding and removing items, and inspecting the front item.

The data in a queue is logically organized as a linear sequence of cells.

Thus each item added to a queue has a position in the queue corresponding to the size of the queue (length) at the time the item was enqueued.

The size of the queue is the difference between the number of *enqueue* and *dequeue* operations starting from an empty queue.

The FIFO order of the queue makes it useful for many operations in which the least recent item should be accessed first.

Note

Applications of the queue ADT:

- computer resource scheduling
- printer spooling
- iterative algorithms, e.g. Breadth-First Search

©2017 UMBC Training Centers

Example Queue ADT

Let's consider a fictional job scheduling application to exercise the queue interface.

The premise is we have a supercomputer that will process jobs as they come in.

But alas, the supercomputer has a finite amount of resources. Thus if the incoming rate of jobs is greater than the rate of processing jobs, then the jobs will backlog (queue up).

We will design a job scheduler for the supercomputer so it can manage the queue of jobs.

Our job scheduler must allocate jobs to compute resources. Each job is an object that contains meta-data about the minimum number of CPUs and amount of RAM it requires.

Our job scheduler holds a queue of fixed-size on which it adds at most *n* jobs. The queue takes data of the job type.

But at close-of-business (COB) the supercomputer must be shutdown to rest, and thus the queue is emptied for the next day.

Note

A little law dictates the size of a queue given the arrival rate versus the service time (surprisingly no other information is needed).

Let λ be the average arrival rate and W be the average service (wait) time. Then the average length L of the queue is simply: $L = \lambda W$.

This is known as Little's Law, named after John Little who published it in 1954.

Queue ADT - job scheduler

Since we are using a queue ADT to design our job scheduler, we can leave out many messy details about real-world supercomputers and applications. It also gives us the flexibility in the design of our algorithm.

This is the purpose of abstract data types.

For example, since we are using an abstract queue, we can assume it is lock-free and cannot get race conditions (surely, some expert could build such a thing).

Moreover, our job scheduler can get jobs from some input and in parallel, manage the allocation and deallocation of jobs from the queue (surely, this is also possible in real-life).

Our job scheduling algorithm using the queue interface is given next. In the algorithm given below, assume that Q is an abstract ADT and is thusly lock free (it will not allow for race conditions). Additionally, assume that it takes at most one minute to empty the queue, and at

©2017 UMBC Training Centers

the start and end of day, the queue must be empty. Suppose that we start the supercomputer and job scheduler at 0500 and shut it down at 1700.

```
Require: Q                                                  ▷ A queue.
Require: A clock                                   ▷ Gets the time of day.
Require: A supercomputer.
  while time $<$ 1659 do              ▷ Asynchronous loop and interrupt
    for all jobs $J$ input by graduate students do    ▷ Parallel region
      if Q size $< n$ then
        enqueue $J$ onto Q.
      else
        wait until Q has room then enqueue $J$ onto Q.
    while Q is not empty do                           ▷ Parallel region
      if not holding a $j$ from last dequeue then
        set $j$ := job from dequeue
      else
        if fifteenth attempt to schedule $j$ then
          send $j$ back to owner.
          set $j$ := job from dequeue
      while wait time is less than two minutes do
        if available supercomputer resources to service $j$ then
          send $j$ to supercomputer with allocated CPU/RAM.
          stop waiting
      if still holding $j$ then
        if Q size $< n$ then
          enqueue $j$ onto Q.
  while Q is not empty do
    dequeue job
```

Exercise Set 4: Stacks and Queues

Utilizing `structs`, you will be creating a new data structure: the **self-aware stack** and **self-aware queue**.

Using the principles of amortized analysis, we will increase /decrease the size of the stack and queue based on parameters provided when running the program. Assume that the variables x and k will be provided as a `float` and an `integer`, respectively.

The stack will begin at size s. As items are `push`ed onto the stack, it will fill up. Once it reaches $x\%$ capacity, you are required to increase its size by a factor of k. For example, if $x = 0.5$ and $k = 2$, you would double the size of the memory allocated to the stack, while preserving the elements within. You will continue to perform this expansion as long as the expansion criteria are met, until the stack has, or exceeds a capacity of $128s$. Similarly, when the occupancy of the stack drops below x, you are expected to shrink the stack's capacity at the same rate until it once again returns to s.

Implement a queue with the exact same capacity constraints/resizing policy.

Expect that the combinations of s, x, and k that you are to deal with will not cause issues.

You are expected to implement a stack and a queue, both of which confirm to the above capacity and resizing policies. Assume that s, x, and k are provided, and will be parameters for your `init_stack` and `init_queue` functions.

Implement the following functions for your queue.

a) Enqueue

b) Dequeue

c) Capacity (current capacity)

d) Num_Elements (return number of elements currently in data structure)

Implement the following functions for your stack.

a) Push

b) Pop

c) Capacity (current capacity)

d) Num_Elements (return number of elements currently in data structure)

Aside: Amortized Analysis

Resizing of data structures at the right time is paramount to memory management and time complexity efficiency in computer science! In this exercise, you have the ability to customize the implementation of your resize policy. For further reading, explore **amortized analysis of algorithms**.

©2017 UMBC Training Centers

Chapter 13

Sorting

Sorting is a fundamental task in the organization and analysis of data.

It is often employed in the design and analysis of algorithms, and for many algorithms their time complexity depends upon the performance of sorting.

At a basic level, we sort information because we want to start from the beginning and progress to the end, and know how long it may take.

This implies some principle of ordering. If we had a well-ordered set then we know that there is some least element, and when comparing two elements we know which should precede the other.

It is much easier to find an element among others if the elements were in an ordered sequence rather than in a disarray of random order.

Thus, sorting facilitates searching.

How does one sort a collection of elements?

Given n elements and picking each in random fashion leads to $n!$ possible permutations of which only one is of interest.

The performance of a sorting algorithm is therefore important for timeliness concerns.

©2017 UMBC Training Centers

Description

It should be evident that picking items at random is not a timely method for sorting a set of items.

Given n items there are $n!$ permutations. Suppose $n = 20$ and a permutation is generated every trillionth of second. It would still take 28 days to get all permutations.

Note

If $n = 30$ it would take over 3,070,056,247,826,285 days or about 8 trillion years.

The age of the universe is about 13.8 billion years.

Note that a sorted sequence of the n items is itself a permutation. Thus any permutation of a set of items is an ordering of the items.

Given the number of permutations, one can ask if there is limit to how fast a set of items can be sorted.

There is in fact a provable lower-limit on sorting performance.

©2017 UMBC Training Centers

Lower Bound

Any deterministic algorithm that sorts by comparing elements takes $\Omega(n \log n)$ time for n items — such an algorithm makes $\Omega(n \log n)$ comparisons.

Given an input $x_1, x_2, \ldots, x_n$ there is a permutation of that input that gives the correct sort order. For example:

input:	$[3, 4, 2, 1]$
output:	$[x_4, x_3, x_1, x_2] = [1, 2, 3, 4]$

There are $n!$ possible inputs from a set of n distinct elements. Then there must be a permutation for each input that sorts it.

Any sorting algorithm must be able to produce all $n!$ permutations.

An algorithm can of course do more than just compare elements, but it must make some comparisons and each comparison should account for a single pair of elements, meaning it cannot be some super comparison.

Let's set some constraints for sorting by comparing elements so the $\Omega(n \log n)$ lower-bound applies to any comparison-based sorting algorithm that can ever be conceived.

i) Each input of distinct elements has one correct output permutation.
ii) Only comparisons between two elements can lead to a decision,

©2017 UMBC Training Centers

e.g. is x_i less than x_j?

Now we'll give the intuition behind the proof on lower-bound performance of sorting.

Observe that each comparison is binary, resulting in two outcomes.

Either x_i is less than x_j or not — either true or false.

Each outcome leads to another comparison, stopping when no further comparisons can be made.

Thus after k steps there are 2^k outcomes.

Since the algorithm must produce $n!$ outcomes, then $2^k \geq n!$ implies $k \geq \log(n!) = \Omega(n \log n)$.

©2017 UMBC Training Centers

This page intentionally left blank.

CHAPTER 14

Lower-bound

Any deterministic, comparison-based sorting algorithm takes $\Omega(n \log n)$ time to sort n items.

This holds for any algorithm that has been conceived and will ever be conceived.

Thus sorting by comparing elements, at best is slower than linear time.

This astonishing claim has a relatively simple proof.

The proof uses a decision tree model.

Decision tree model

The decisions made by any sorting algorithm after comparing two elements can be modeled by a decision tree.

Each internal node in the tree represents a comparison between a pair of elements.

A leaf node gives the sorted order for an input.

Label a node $i < j$ and let the left branch be the path taken if $x_i < x_j$ and the right branch if $x_i \geq x_j$.

A path from root to a leaf is the sequence of comparisons that gives the correct permutation for one of the $n!$ inputs.

Then the maximum number of comparisons correspond to the height of the tree.

A binary tree of height h has at most 2^h leaves.

Since there must be at least $n!$ leaves, this implies

©2017 UMBC Training Centers

$$
\begin{aligned}
2^h &\geq n! \\
h &\geq \log(n!) \\
&\geq \log\Big((\frac{n}{e})^n\Big) && \text{(by Stirling's Approximation)} \\
&= n \log n - n \log e \\
h &= \Omega(n \log n).
\end{aligned}
$$

Therefore any algorithm that sorts by comparing elements takes $\Omega(n \log n)$ time.

Illustration

Consider $n = 3$ and any decision that has a true outcome leads to the left branch, otherwise it leads to the right branch.

- left branch $\equiv$ true
- right branch $\equiv$ false

Figure 14.1 illustrates the decision tree model for any comparison-based sorting algorithm this simple set of inputs.

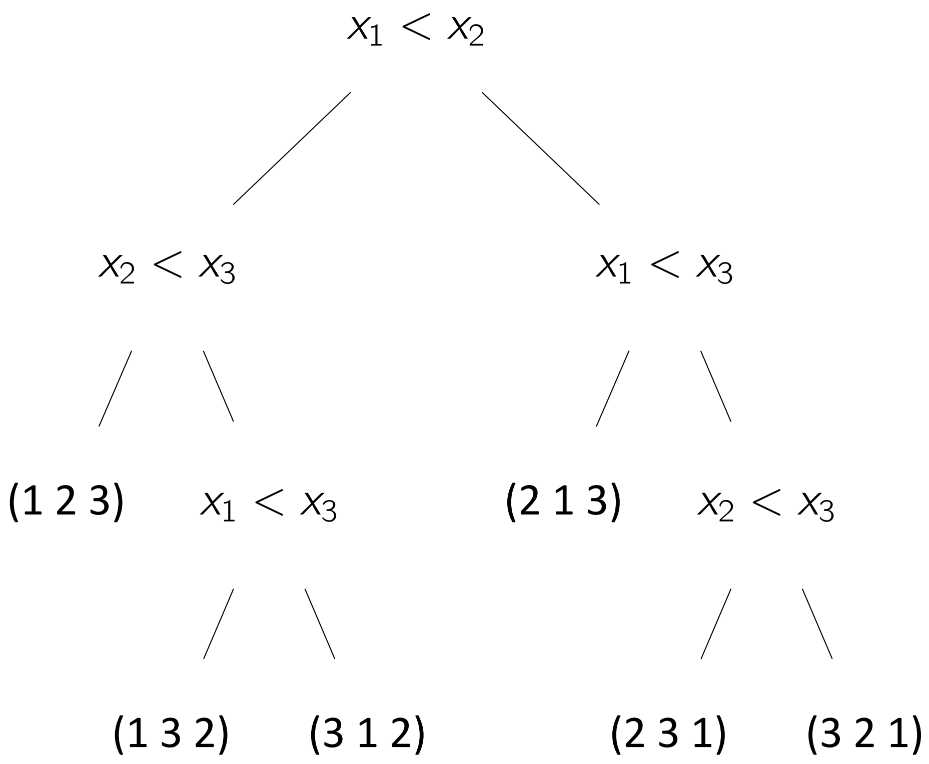


Figure 14.1: Decision tree example for comparison-based sorting.

©2017 UMBC Training Centers

This page intentionally left blank.

©2017 UMBC Training Centers

Chapter 15

Insertion Sort

Insertion Sort

A comparison-based **sorting algorithm**.

It is an iterative algorithm that maintains an intermediate sorted sequence by taking each item in the input, placing it in order, until the sequence is completed.

It takes $O(n^2)$ time for both the average and worst case.

It sorts *in-place* and therefore does not require additional space.

Description

The insertion sort algorithm is very simple and despite taking $O(n^2)$ time for average and worst cases, in practice it performs well.

It has the following advantages:

- Easy to implement.
- Sorts in-place; requires no additional space.
- Stable, meaning the relative order of duplicate items is preserved.
- Can sort on a stream as items are added.

On small input the algorithm is relatively fast and efficient. If the input is already sorted or nearly sorted, it takes $O(n)$ time and this is the best case.

But the algorithm does poorly on very large input, taking quadratic time.

©2017 UMBC Training Centers

Algorithm

The insertion sort algorithm iteratively builds an intermediate sorted sequence by taking each item from the input and places it in order by exchanging with items already in order.

Given some input L, the first item in L is the first item in the intermediate sorted sequence. In the end L will be sorted.

The algorithm reads the next input item i and then working towards the front of L, it compares i to the items j in L and exchanges places with each j until it finds a j such that $j \leq i$, otherwise i is then at the front of L. A position counter is maintained so each iteration step gets a new input item.

The iteration completes until all input items have been consumed so then L is the final sorted sequence.

The basic premise of this procedure is a sorted sequence is grown incrementally where inverted items are exchanged. Inverted means that items are out of order.

An insertion sort algorithm that takes an array and works from front to back, using zero-indexing, is given next.

An algorithm working from back to front is as follows.

©2017 UMBC Training Centers

```
Require: A                                          ▷ an array of size $n$
for $i = 1$ up to $n$ do
    set $j$ := $i$
    while $A[j-1] > A[j]$ and $j > 0$ do
        exchange $A[j-1], A[j]$
        set $j$ := $j - 1$
```

```
Require: A                                          ▷ an array of size $n$
for $i = n - 2$ down to $0$ do
    set $j$ := $i$
    while $A[j] > A[j+1]$ and $j < n - 1$ do
        exchange $A[j], A[j+1]$
        set $j$ := $j + 1$
```

Time complexity

It isn't difficult to see that the iteration can repeatedly compare and exchange each item in L.

Given an item at position p, it is possible that it must be exchanged with all $p-1$ previous items placed in partial order. Then for n items the work is,

$$1+2+3+\ldots+n-1=\sum_{i=0}^{n-1} i=\frac{n(n-1)}{2}=O(n^2).$$

Therefore in the worst case insertion sort takes $O(n^2)$ time.

The worst case is possible given a sorted input in descending order. The algorithm effectively reverses the input to ascending order.

Conversely, if given a sorted sequence (or nearly sorted) then it simply adds each input item to the end of L and therefore takes $O(n)$ time.

The runtime on average is $O(n^2)$ time, which can be proved by summing the expectation over all pairs of X_{ij} random variables that equal one if there is inversion and zero otherwise, namely indicator variables. We leave the proof as an exercise for the interested reader.

©2017 UMBC Training Centers

Chapter 16

Quick Sort

Quick Sort

A comparison-based **sorting algorithm**.

This is a divide-and-conquer type algorithm.

It recursively partitions a set of elements by choosing a pivot and comparing left and right partitions against the pivot.

In the worst case it takes $O(n^2)$ time but on average it takes $O(n \log n)$ time.

It sorts *in-place* and therefore does not require additional space.

The *quicksort* algorithm was invented by Sir Charles Anthony Richard Hoare (Tony Hoare) in 1959.

Description

The quicksort algorithm is quite simple in its design but how it achieves its renowned average-case performance is remarkable.

The algorithm was invented in 1959 but it was not clear how to analyze it and a proof of its average-case came much later.

This is due in part in how to count the number of comparisons made.

In simple prose, the quicksort algorithm is as follows.

> Given an array to sort, choose a pivot from the array.
>
> Now partition the array into two subarrays, L and R, so all elements less than the pivot are in L and all others are in R.
>
> Return the position of the pivot that separates L and R, and recurse until each partition has one or less elements.

Popularity

Despite taking $O(n^2)$ time in the worst-case, in practice the quicksort algorithm is very fast.

It has been proven to have $O(n \log n)$ average-case time and this bears out in many real-world datasets.

But other sorting algorithms have tight bounds of $\Theta(n \log n)$ time, such as *merge sort* and *heap sort*. Meaning these other sorting algorithms have the same asymptotic complexity for any input, whereas quicksort can take quadratic-time in the worst-case.

An analysis of the average time of quicksort reveals that the constant factor hidden by the Big-Oh notation is quite small (approx. 1.39).

This small constant factor, sorting in-place, and cache-friendliness makes quicksort a favorite choice for many real applications.

The quicksort algorithm has been a mainstay choice for sorting for many years. It is the algorithm implemented in the C library *qsort* function.

Partitioning

Over the years there have been a number of popular partitioning schemes in addition to the original Hoare partition.

The goal of partitioning is to maintain an even division of work at each step of the recursion.

In the best case all subpartitions are half the size of the previous subpartitions.

The Hoare partition is still a good choice for many applications.

Another partitioning scheme is due to that of Nico Lomuto.

The Lomuto partitioning popularized was in the book, “Programming Pearls”, by Jon Bentley, and in the computer science textbook, “Introduction to Algorithms”, by Cormen et. al.

The Lomuto partitioning is considered by some to be easier to implement, but it performs more comparisons in practice than the Hoare partitioning.

We’ll describe both Hoare and Lomuto partitioning.

Note

Other partitioning schemes in addition to the original Hoare partitioning are:

- Lomuto
- median-of-three
- Bentley-McIlroy 3-way
- cutoff-to-insertion sort

Hoare Partition

Use two pointers at opposite ends of the array.

The *left* pointer starts at the beginning and moves right until it finds an element greater than the pivot.

The *right* pointer starts at the end and moves left until it finds an element less than the pivot.

When the pointers have stopped, exchange the two elements if the *left* pointer is still left of the *right* pointer.

Repeat until the two pointers cross, hence the two partitions are correctly filled.

The *right* pointer now marks the end of the left partition and the beginning of the right partition.

Algorithm - Hoare

The quicksort recursive algorithm using the Hoare partition is described next.

The Hoare partitioning function is given first followed by the main quicksort recursive algorithm.

```
Require: A                                    ▷ an array of size $n$
function PARTITION(i, j, A)
    set $p$ := $A[i]$
    set $l$ := $i - 1$
    set $r$ := $j + 1$
    while true do
        repeat
            set $l$ := $l + 1$
        until $A[l] < p$
        repeat
            set $r$ := $r - 1$
        until $A[r] > p$
        if $l < r$ then
            exchange $A[l], A[r]$
        else
            return $r$
```

Here is the recursive quicksort algorithm.

```
Require: A                                        ▷ an array of size $n$
function QUICKSORT(i, j, A)
    if $i < j$ then
        set p := PARTITION(i, j, A)
        QUICKSORT(i, p, A)
        QUICKSORT(p+1, j, A)
```

Lomuto Partition

For each subarray $A[i] \ldots A[j]$, choose the pivot to be the last element of the subarray.

Start pointers l, r at the beginning of the subarray.

Move r right until an element is less than the pivot, at which point $A[l]$, $A[r]$ are exchanged and l is moved one position to the right.

Repeat until r reaches the end of the subarray, then exchange $A[l]$, $A[j]$ and return l as the new position that separates the left and right partitions.

©2017 UMBC Training Centers

Algorithm - Lomuto

The quicksort recursive algorithm using the Lomuto partition is described next.

The Lomuto partitioning function is given first followed by the main quicksort recursive algorithm.

```
Require: A                                              ▷ an array of size $n$
function PARTITION(i, j, A)
    set $p$ := $A[j]$
    set $l$ := $i$
    set $r$ := $i$
    while $r < j$ do
        if $A[r] < p$ then
            exchange $A[l], A[r]$
            set $l$ := $l + 1$
        set $r$ := $r + 1$
    exchange $A[l], A[j]$
    return $l$
```

Here is the recursive quicksort algorithm.

```
Require: A                                              ▷ an array of size $n$
function QUICKSORT(i, j, A)
    if $i < j$ then
        set p := PARTITION(i, j, A)
        QUICKSORT(i, p-1, A)
        QUICKSORT(p+1, j, A)
```

©2017 UMBC Training Centers

Time Complexity - Best case

If the pivot is always the median, then each subpartition is half the size of the previous.

This comparable to the subdivision in *mergesort*.

Thus there are at most $\frac{n}{2}$ elements in each partition in the first step, and then half this in the next partition, continuing until one element remains in a partition.

This subdivision follows,

$$\frac{n}{2}, \frac{n}{2^2}, \frac{n}{2^3}, \dots \frac{n}{2^k}.$$

It takes $\Theta(n)$ work to create a partition since the pointers move through half the input.

Another way to think of this is $n - 1$ elements are compared to the pivot.

This leads to the recurrence relation,

$$T(n) = 2T(\frac{n}{2}) + n.$$

By the Master Theorem, $T(n) = \Theta(n \log n)$. It also easily proved by mathematical induction.

©2017 UMBC Training Centers

Time Complexity - Worst case

If the pivot is always the minimum or maximum value, then one partition is empty and the other has all remaining elements.

Then each subdivision is one less in size than the previous so it will take $n-1$ steps. Thus the work follows,

$$\begin{aligned}(n-1)+(n-2)+\ldots+1 &= 1+2+\ldots+(n-1)\\ &= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}\\ &= O(n^2).\end{aligned}$$

The intuition behind this is each element can be the pivot only once, thus each element is compared to all other elements once, leading to $\frac{n(n-1)}{2} = \binom{n}{2} = O(n^2)$ comparisons.

The recurrence relation is then $T(n) = T(n-1) + n$.

This because it takes $\Theta(n)$ to create the first partitions, and then at each step the partition decreases in size by one with the other partition being empty.

It is easy to prove by induction that $T(n) = T(n-1) + n = O(n^2)$.

Time Complexity - Average case

After choosing a pivot from 1 to n distinct elements there will be two partitions of size i and $n - i - 1$.

It takes $n - 1$ comparisons to create the partitions which leads to,

$$T(n) = T(i) + T(n - i - 1) + (n - 1).$$

Any of the n elements is equally likely to be the pivot.

The worst case is one partition is empty and the other has $n - 1$ elements.

Averaging over all possible pivots and resulting partitions gives,

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} \Big(T(i) + T(n - i - 1) \Big) + (n - 1).$$

Observe that $T(i)$ and $T(n - i - 1)$ sum the same, only one counts up and the other down. Hence,

$$T(n) = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n - 1.$$

Multiplying by n to both sides gives,

$$nT(n) = 2\sum_{i=0}^{n-1} T(i) + n(n-1).$$

We can use this relation for n and $n-1$ and get the difference,

$$\begin{aligned}
nT(n) - (n-1)T(n-1) &= 2T(n-1) + n(n-1) - (n-1)(n-2) \\
&= 2T(n-1) + 2(n-1) \\
nT(n) &= 2T(n-1) + (n-1)T(n-1) + 2(n-1) \\
&= T(n-1)(2+n-1) + 2(n-1) \\
&= (n+1)T(n-1) + 2(n-1).
\end{aligned}$$

This is leads to a new recurrence,

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \\
&= \sum_{i=0}^{n-1} \frac{2(i-1)}{i(i+1)} \\
&\approx 2\sum_{i}^{n} \frac{1}{i} \approx 2H_n \approx 2\ln n && \text{(Harmonic series)}
\end{aligned}$$

Thus $T(n) \approx 2(n+1)\ln n \approx 1.39 n \log n = O(n \log n)$.

©2017 UMBC Training Centers

Exercise Set 5: Sorting

a) Implement insertionsort as it is described in the textbook. You will need it for part b as well.

b) Implement the brand-new sorting algorithm: **smart-quicksort**. Suppose you are given an input array of `ints` of size n along with a number $k < n$. Assume that $k|n$. (n will always be a multiple of k).

As you know, quicksort is a recursive algorithm. You will be implementing it as such- perform regular quicksort on the given input array, until the block sizes you are working with are of size n or smaller. At that point, instead of recursively performing quicksort, perform insertionsort. You are encouraged to make use of the insertionsort implementation you created in part a. You are welcome to use any partitioning method discussed in the text.

c) Implement a new algorithm: **goofy-quicksort**. Here, you will be implementing quicksort, except instead of selecting a partition using a given method, you will need to select the **worst case element to partition on** for every recursive iteration of the algorithm. Naturally, to accomplish this, you are allowed extra time and space complexity to determine the worst-case element to partition on.

©2017 UMBC Training Centers

If you wish, you are allowed to use the function you produced in part b to help you **determine the worst element to pivot on**, **not to sort**.

d) Best and worst cases are important for sorting algorithms. It's paramount that aspiring computer science experts understand these. Suppose you are given an array of 100 distinct integers, from 0 to 99.

   i. What order, if any, would these integers need to be in for insertion sort to have best case performance? How about worst case performance?

   ii. What order, if any, would these integers need to be in for quicksort to have best case performance? How about worst case performance? Assume you are using the Lomuto Partitioning schema.

Chapter 17

Tree

A tree is an abstract object that is used in mathematics and computer science to represent data and processes.

It is fundamental to many data structures and algorithms.

The nonlinear and recursive nature of trees will be explored and its importance should become increasingly evident.

Description

In graph theory a tree is a special type of graph that is an abstract representation of a network that contains no cycles.

There are many types of trees including unordered, ordered, free, rooted, oriented, directed, algebraic, etc.

We will begin first with the definition of an unordered tree used in the graph theory setting. We then describe a rooted tree followed by trees induced by a specific ordering.

Throughout this text we will often refer to these objects simply as a tree and let the type be implied by the context.

©2017 UMBC Training Centers

This page intentionally left blank.

CHAPTER 18

Unordered Tree

Tree A collection of linked nodes with no cycles and each node is linked to one or more other nodes.

That is a tree is a set of nodes and a set of edges. An edge connects a pair of nodes.

There exists a path between every pair of nodes. But there is no path through distinct nodes that begin and end at the same node, meaning there is no cycle.

There is one and only one path between a pair of nodes, because there are no cycles.

A tree is therefore a connected, acyclic graph of n vertices and $n - 1$ edges.

Forest

A collection of more than one tree.

©2017 UMBC Training Centers

Description

A few examples of a tree are illustrated in Figure 18.1.

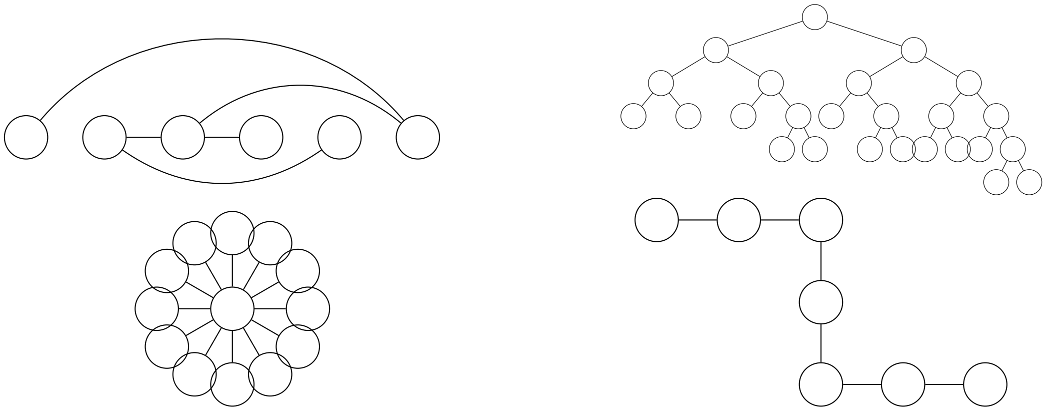


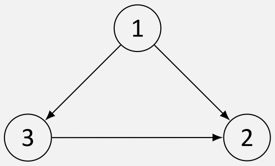
Figure 18.1: Tree Examples

Observe that a tree is an undirected, acyclic graph in which edges have no direction.

A Directed Acyclic Graph (DAG) has no directed cycles, but a DAG should not be confused with a tree. The underlying undirected graph of a DAG may have a cycle.

Note

A DAG cannot be a (directed) tree because there can be more than one directed path between a pair of vertices.



Not a tree! There are two paths from nodes 1 to 2.

Graph Application

Unordered trees are used in graph theory to model network data. The Spanning Trees of a graph are often used in network design.

Note

The Minimum Spanning Tree (MST) is used to find the least cost network that links every vertex in a graph. Applications include:

- minimizing the cost of laying fiber optic cable
- spanning tree protocol for ethernet networks (unweighted MST)
- handwriting recognition
- clustering data

Given a simple, undirected graph, a basic question asks whether or not the graph is a tree.

How do you determine if a graph is a tree?

It is equivalent to detecting if a cycle exists!

©2017 UMBC Training Centers

Summary

The following is a summary of the properties of (unordered) trees.

- A tree has no cycles.
- A path exists between every pair of nodes in the tree.
- Every pair of nodes is connected by one and only one path.
- A tree of n nodes has $n - 1$ edges.
- Edges have no direction.

These properties lead to the following observations (or equivalent properties).

Adding an edge to a tree creates a cycle.

Removing an edge from a tree disconnects it.

Up to this point we have considered a tree as an unordered network of linked nodes that contains no cycles, namely as an undirected, acyclic graph.

But some types of trees have an ordered structure, and can therefore be used model hierarchical data. This suggests that these trees have some starting point.

We'll begin by describing rooted trees next before investigating ordered trees.

Chapter 19

Rooted Tree

Rooted Tree

A **tree** with a single root node from which all other nodes are descendants.

A neighbor of a node is either its parent or child. The parent node precedes a child node from the direction descending from the root node.

Every node can have zero or more child nodes.

The path from the root to any node is unique, because there are no cycles.

It then follows that every node, but the root, can only have a single parent.

©2017 UMBC Training Centers

Description

The basic properties of a tree still hold for a rooted tree. That is a rooted tree cannot have cycles, there are $n - 1$ edges connected n nodes, and a unique path exists between every pair of nodes.

But a rooted tree has a hierarchy that begins with the root node. This leads to a natural notion of direction from and to the root; descending from the root or ascending from a node to the root.

The top of the tree is the root.

The following are definitions for the different nodes of a rooted tree.

root node A node that has no parent and all other nodes descend from it.

internal node Any node having a parent and at least one child node.

leaf node Any node having a parent but no children.

A tree can be rooted at an arbitrary node from which all other nodes are descendants of the root node. See Figure 19.1 for a simple example of a rooted tree.

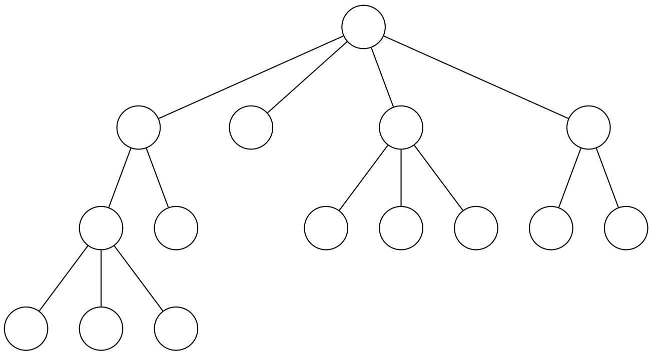


Figure 19.1: Rooted tree

A rooted tree is easily defined recursively because of its hierarchy.

1. Let v be the root node of a tree T, then v is the ancestor of all other nodes in T.

2. For each child node u of v, relabel u as v then recurse at step 1.

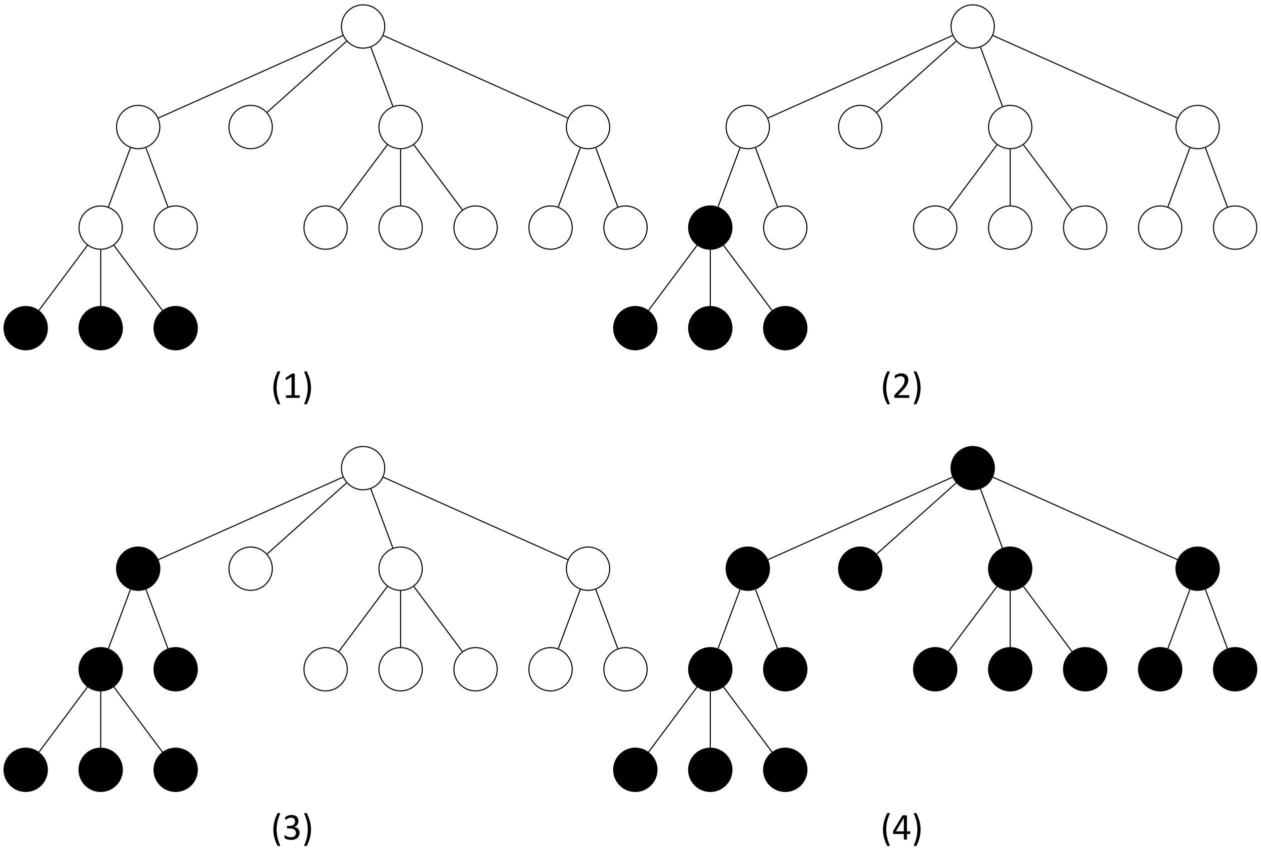
This introduces the concept of subtrees. Each node in a rooted tree is itself the root of a subtree.

Note

All trees can be defined recursively.

Figure 19.2 demonstrates subtrees by filling the nodes of the left subtree moving up to the root.

©2017 UMBC Training Centers



(1)

(2)

(3)

(4)

Figure 19.2: Subtrees

©2017 UMBC Training Centers

Given the descriptions of a rooted tree thus far, there is a natural separation of nodes based on their distance from the root node.

Hence the children of the root are one step away. We call this the first level.

Then the children of these are two steps removed from root, and hence are in the second level. The root is the grandparent of these nodes.

It should be obvious that the nodes in the last level are all leaf nodes.

This leads to two important properties of a rooted tree.

tree height The longest path from the root to a leaf (tree depth).

tree level The stepwise distance from the root.

The root node is at level 0. The children of the root are at level 1, the grandchildren at level 2, and so on.

Let h denote the tree height and l the number of levels. Then the number of levels $l = h + 1$ is one more than the height because levels include the zeroth-step level, i.e. root level.

It follows that the tree height is one less than the maximum tree level.

Level 0 is the *root level*.

Level h is the *last level*.

The path from the root to a node then represents an ancestry chain.

A rooted tree therefore implies a hierarchy and ordering of the nodes. This is the basis for the tree data structure.

©2017 UMBC Training Centers

Before going further, let's look at the tree rooted at node 5 in Figure 19.3.

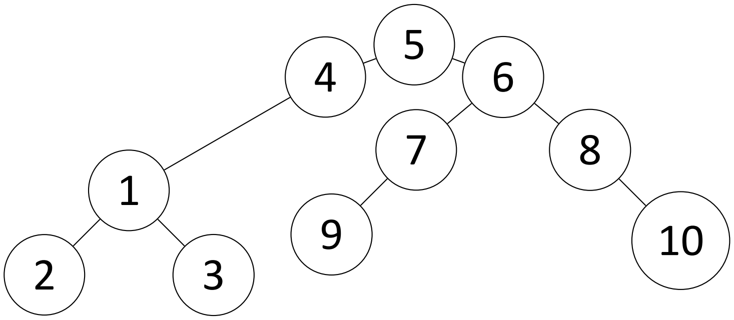


Figure 19.3: Rooted tree (root node: 5)

We can immediately see it has $h = 3$ height and $l = 4$ levels.

Suppose instead the tree is rooted at node 6, as depicted in the next figure.

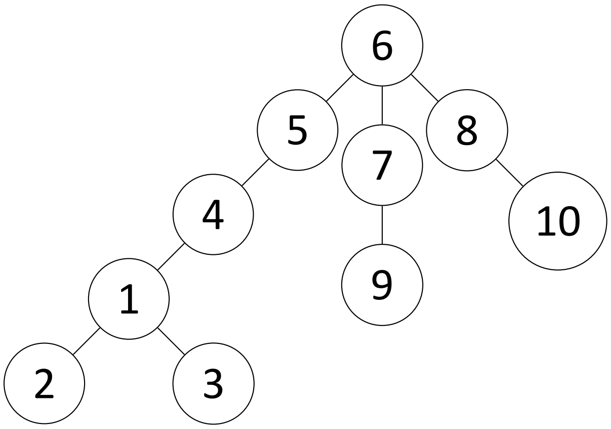


Figure 19.4: Rooted tree (root node: 6)

The maximum number of children in Figure 19.4 is three as opposed to two in Figure 19.3.

Also this tree (Figure 19.4) has $h = 4$ height and $l = 5$ levels.

The two trees in Figures 19.3 and 19.4 are isomorphic; the underlying unordered tree is the same. But the maximum number of children and the height are greater for the tree in Figure 19.4. Although the underlying graph of these rooted trees are the same, the ancestry is very different.

Note

The root node is arbitrary in a rooted tree. Selecting different nodes for the root does not change the underlying (unordered) tree.

Labeled trees

Before moving to the next type of ordered tree, let us consider the use of labels on nodes.

So far we have treated trees simply as a network of nodes, specifically unlabeled nodes and edges without specifying any attributes on the nodes or edges.

Nodes can be distinguished implicitly by their ancestry and because the path between any pair of nodes is unique.

But observe that exchanging any two nodes in a tree leaves the tree indistinguishable from its previous state. This is true for any tree.

Therefore labels or keys can be added to nodes to distinguish nodes and impose specific structure and ordering.

Trees are used to represent data or operations so it is natural to use a label or key for data elements. Often it is the data itself that is the label.

We will use key, label, and value interchangeably and where appropriate will distinguish differences in usage.

Also, we were refer to the nodes of tree by its key or label, e.g. node i means a node whose key is i.

Later we will see that node keys are necessary for imposing ordering

©2017 UMBC Training Centers

rules on every node.

k-ary Tree

k-ary Tree

A **rooted tree** in which each node has at most k child nodes, where k is a constant.

Note

It is also known as a m-ary tree; each node has at most m children.

Thus the tree in Figure 19.3 is a 2-ary tree but when rooted at node 6 it becomes a 3-ary tree in Figure 19.4, or also known respectively as binary and ternary tree.

The trees we have covered until now had no restriction on the number of children, or branches, from a parent node. But the limit on the maximum number of child nodes make it easier to analyze the shape and size of the tree. Specifically we can determine the upper-bound on nodes in each level, the total size overall, and a range for the height of the tree.

©2017 UMBC Training Centers

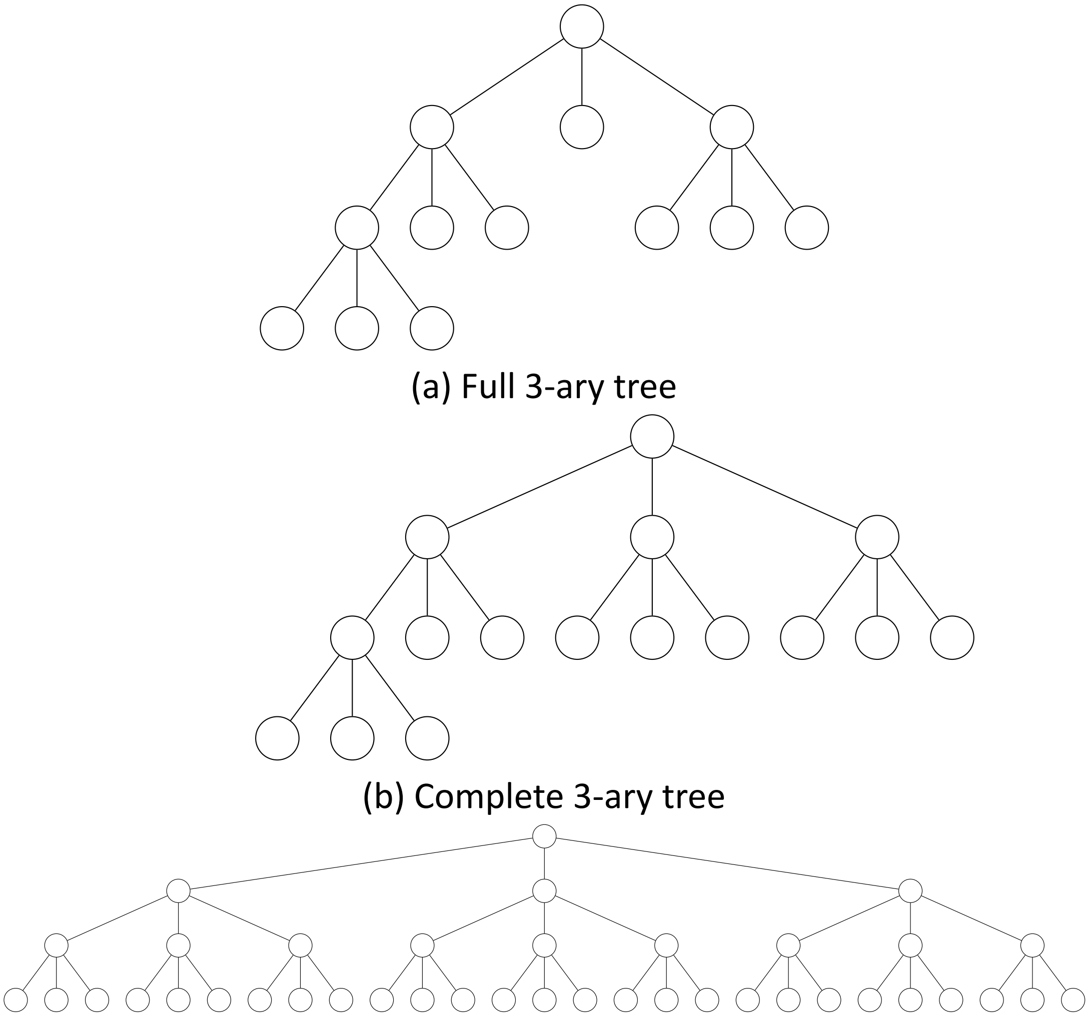
The following are definitions for k-ary trees based on the fullness or completeness of the levels.

full k-ary tree
Each node has 0 or k children.

complete k-ary tree
Every level but the last must be maximally filled, i.e. complete. The last level must be filled from left-to-right and can also be complete.

perfect k-ary tree
All levels are maximally filled.

Examples of these k-ary trees are illustrated in Figure 19.5 for $k = 3$.



(a) Full 3-ary tree

(b) Complete 3-ary tree

(c) Perfect 3-ary tree

Figure 19.5: k-ary trees ($k = 3$)

©2017 UMBC Training Centers

A k-ary tree has one node at level 0. The maximum number of nodes at level 1 is then k. At level 2 each of the k children can have at most k children and so the maximum size is k^2. This progresses geometrically so the leaf level h has k^h nodes.

> The tree height h is the maximum depth.
>
> ---
>
> Then $l = h + 1$ is the number of levels.

Hence any i^{th} level of a k-ary tree has at most k^i nodes.

©2017 UMBC Training Centers

The upper-bound on the size of a k-ary tree depends on its height.

Each level i of a k-ary tree has at most k^i nodes. The maximum number of nodes over all the levels is then,

$$\sum_{i=0}^{h} k^i = \frac{k^{h+1}-1}{k-1}.$$

Note

Recall the sum of a geometric series.

$$1 + x + x^2 + x^3 + \ldots + x^{n-1} = \sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

Since $l = h + 1$ we can also write the expression as, $\sum_{i=0}^{h} k^i = \frac{k^l-1}{k-1}$.

Then a k-ary tree has $n \leq \frac{k^{h+1}-1}{k-1}$ nodes. The asymptotic upper-bound for n is then,

$$\begin{aligned}
\lim_{k\to\infty} \frac{k^{h+1}-1}{k-1} &= \lim_{k\to\infty} \sum_{i=0}^{h} k^i \\
&= \lim_{k\to\infty} k^h + k^{h-1} + \ldots + k + 1 \\
&= k^h \\
&= O(k^h).
\end{aligned}$$

©2017 UMBC Training Centers

Hence there are $O(k^h)$ nodes in a k-ary tree.

©2017 UMBC Training Centers

The tree height is a parameter in the upper-bound on the number of nodes in a k-ary tree. Using this upper-bound on the size of the tree we can get a lower-bound on the height h.

Let n be the total number of nodes in a k-ary tree. Recall the upper-bound of n is,

$$n \leq \lim_{k \to \infty} \frac{k^l - 1}{k - 1} = k^l = O(k^l).$$

We use the number of levels l for convenience since h bounded by l. The lower-bound is then,

$$\begin{aligned} k^l &\geq n \\ l &\geq \log_k n \\ &= \Omega(\log n). \end{aligned}$$

It follows that $h = \Omega(\log n)$ is the minimum height for a k-ary tree of n nodes.

Note

Recall log conversion.

$$\log_a n = \frac{\log_b n}{\log_b a}$$

When a, b are constants then $\log_b a$ is also a constant. Then in the limit of $n \to \infty$ the log bases can be dropped.

©2017 UMBC Training Centers

We have found that $h = \Omega(\log n)$ is the lower-bound for the height h of a k-ary tree of n nodes.

What is the upper-bound on the height?

Suppose now that each node has exactly one child. The tree is then just a path of n nodes. There are at most n nodes so the length of this path is the upper-bound on the height. Then in the limit, the height h ranges in the interval,

$$\log_k n \leq h \leq n.$$

Thus $h = O(n)$ is the upper-bound height of a k-ary tree of n nodes.

The number of nodes at any level i of a k-ary tree is k^i, and for height h the total number of nodes n is bounded by k^h. It is useful to relate these to the fraction of nodes at each level in terms of both n and powers of k.

Recall the geometric sum for the nodes in each level leading to $n \leq k^h$.

$$k^0 + k^1 + k^2 + \ldots + k^h = \frac{k^{h+1} - 1}{k - 1}$$
$$\lim_{k \to \infty} \frac{k^{h+1} - 1}{k - 1} = \frac{k^{h+1}}{k} = k^h$$

The maximum number of nodes in each level progresses as $1, k, k^2, \ldots, k^h$ for l levels. This is equivalent to,

$$\frac{k^h}{k^h}, \frac{k^h}{k^{h-1}}, \frac{k^h}{k^{h-2}}, \ldots, \frac{k^h}{k^0}.$$

Since $n \leq k^h$, then the fraction in terms of n at each level beginning from the root to the last level is,

$$\left\lceil \frac{n}{k^h} \right\rceil, \left\lceil \frac{n}{k^{h-1}} \right\rceil, \left\lceil \frac{n}{k^{h-2}} \right\rceil, \ldots, \left\lceil \frac{n}{k^0} \right\rceil.$$

©2017 UMBC Training Centers

k-ary Summary

The following summarizes the properties of a k-ary tree.

- Each level i of a k-ary tree has at most k^i nodes.

- The maximum number of nodes in each level progresses as the geometric series,

$$k^0, k^1, k^2, \ldots, k^h.$$

- The maximum number of nodes over all levels is,

$$\sum_{i=0}^{h} k^i = \frac{k^{h+1}-1}{k-1} = O(k^h).$$

- The height ranges in the interval,

$$\Omega(\log_k n) \leq h \leq n.$$

- The fraction in terms of n at each level beginning from root to leaf level is,

$$\left\lceil \frac{n}{k^h} \right\rceil, \left\lceil \frac{n}{k^{h-1}} \right\rceil, \left\lceil \frac{n}{k^{h-2}} \right\rceil, \ldots, \left\lceil \frac{n}{k^0} \right\rceil.$$

©2017 UMBC Training Centers

Binary Tree

Binary Tree

A k**-ary tree** where $k = 2$.

Thus a binary tree is a 2-ary tree.

The binary tree is the simplest k-ary tree and is the basis for many results in computer science, particularly because of the logarithmic bounds on size and height.

A binary tree naturally invokes a “left” and “right” part.

It is common to refer a left- or right-subtree, and the children of a node as the left- and right-child.

If there is only one child, is it a left or right child?

This may depend on the order in which children are added!

A binary tree is illustrated in Figure 19.6.

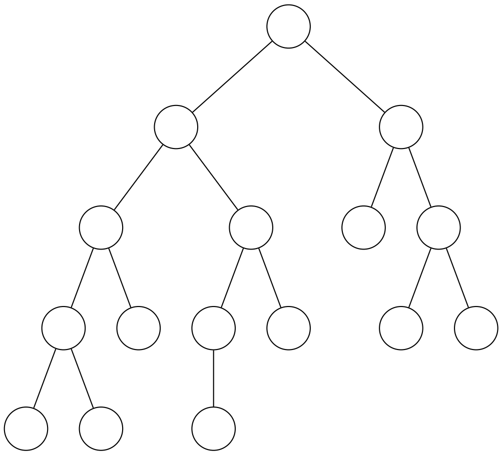


Figure 19.6: Binary Tree

©2017 UMBC Training Centers

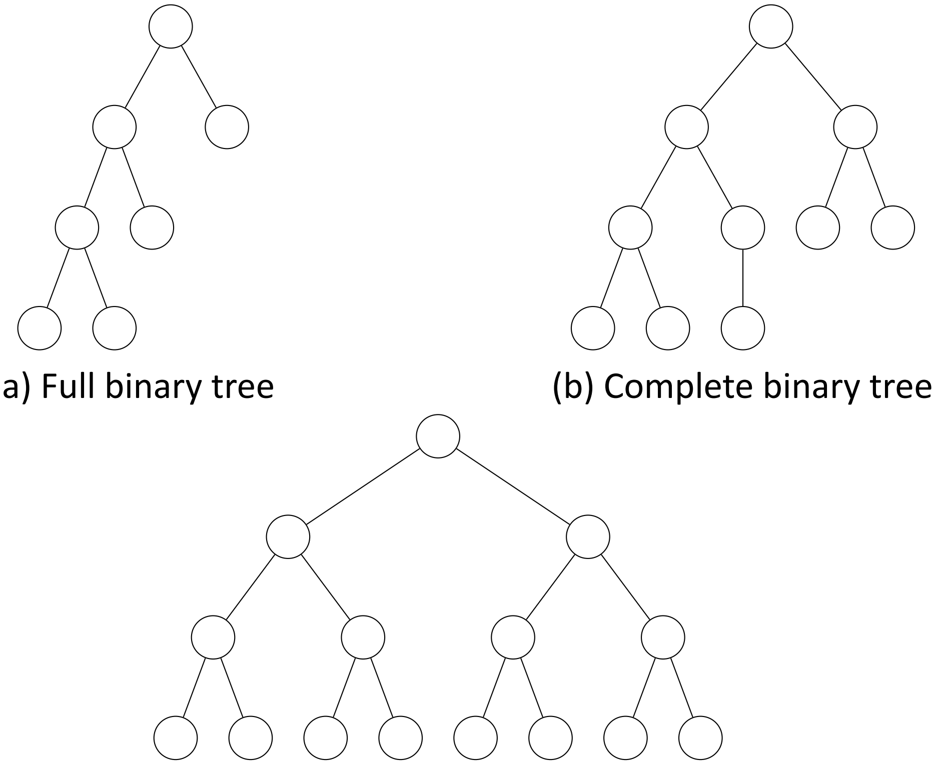
The definitions of binary trees based on the completeness of levels carry over from k-ary trees.

full binary tree
Each node has 0 or 2 children.

complete binary tree
Every level but the last must be maximally filled, i.e. complete. The last level must be filled from left-to-right and can also be complete.

perfect binary tree
All levels are maximally filled.

Examples of these binary trees are illustrated in Figure 19.7.



(a) Full binary tree

(b) Complete binary tree

(c) Perfect binary tree

Figure 19.7: Binary trees

©2017 UMBC Training Centers

All definitions and properties of a k-ary tree hold for a binary tree.

It isn't difficult to see that these properties are independent of an explicit value of k. Simply substituting $k = 2$ gives the specific version of these properties for a binary tree.

k-ary tree properties

1. Each level i of a k-ary tree has at most k^i nodes.
2. The maximum number of nodes in each level progresses as the geometric series,
$$k^0, k^1, k^2, \ldots, k^h.$$
3. The maximum number of nodes over all levels is,
$$\sum_{i=0}^{h} k^i = \frac{k^{h+1}-1}{k-1} = O(k^h).$$
4. The height ranges in the interval,
$$\Omega(\log_k n) \leq h \leq n.$$
5. The fraction in terms of n at each level beginning from root to leaf level is,
$$\left\lceil \frac{n}{k^h} \right\rceil, \left\lceil \frac{n}{k^{h-1}} \right\rceil, \left\lceil \frac{n}{k^{h-2}} \right\rceil, \ldots, \left\lceil \frac{n}{k^0} \right\rceil.$$

binary tree properties

1. Each level i of a binary tree has at most 2^i nodes.
2. The maximum number of nodes in each level progresses as the geometric series,
$$2^0, 2^1, 2^2, \ldots, 2^h.$$
3. The maximum number of nodes over all levels is,
$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 = O(2^h).$$
4. The height ranges in the interval,
$$\Omega(\log_2 n) \leq h \leq n.$$
5. The fraction in terms of n at each level beginning from root to leaf level is,
$$\left\lceil \frac{n}{2^h} \right\rceil, \left\lceil \frac{n}{2^{h-1}} \right\rceil, \left\lceil \frac{n}{2^{h-2}} \right\rceil, \ldots, \left\lceil \frac{n}{2^0} \right\rceil.$$

Using $k = 2$ simplifies many of the properties. The upper-bound on the number of nodes in a binary tree is just $2^{h+1} - 1$ following $\frac{k^{h+1}-1}{k-1}$. The bound on height is given next.

$$\begin{aligned} n &\leq 2^{h+1} - 1 \\ n + 1 &\leq 2^{h+1} \\ \log(n+1) &\leq h + 1. \end{aligned}$$

This implies,

$$h \geq \log(n+1) - 1 \geq \log n.$$

One other interesting result is the last level of a perfect binary tree accounts for half the number of nodes in the entire tree.

Level h has at most 2^h nodes.

There are at most $2^{h+1} - 1$ nodes in total, about twice the number in level h.

It then follows that there are $\lfloor \frac{n}{2} \rfloor$ internal nodes in a complete binary tree.

A complete binary tree can be compactly stored in an array because the nodes can be aligned sequentially in left-to-right order, level-by-level, without gaps.

Note

Any complete k-ary tree can be stored compactly in an array.

This is easily seen by simply labeling the nodes in the tree sequentially from left-to-right, working from top to bottom. See Figure 19.8 for an example. The labels are then the indices in the array and there are no gaps.

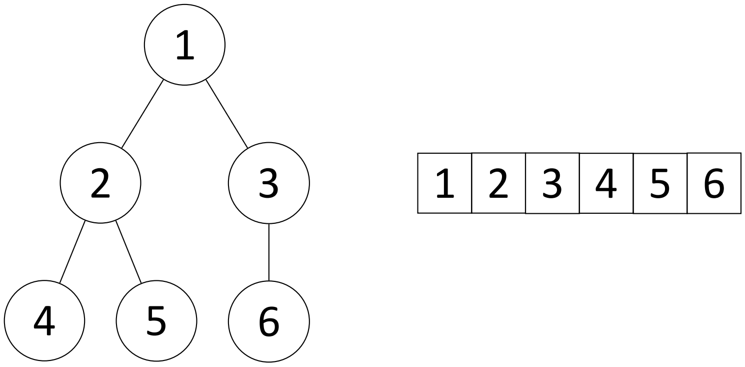


Figure 19.8: Binary Tree stored in array

Observe that the child nodes of a parent must be in the next level of the tree. Every level but the last must be complete, so each complete level has twice as many nodes as the previous. Therefore the children of each node is doubly offset from its parent.

Hence a node at index i must have children at indices $2i$ and $2i + 1$.

©2017 UMBC Training Centers

Conversely, a node at index i has a parent at index $\lfloor \frac{i}{2} \rfloor$.

If using zero-indexing, then the children offsets are $2i + 1, 2i + 2$, respectively for the left and right child nodes, and the parent offset for a node at index i index $\lfloor \frac{i-1}{2} \rfloor$.

A complete binary tree is then stored compactly in an array, having no gaps between indices. This makes it appealing for applications where contiguous memory storage offered by arrays is needed for performance and space efficiency.

Note

Array storage of a complete binary tree is used in Heapsort.

The binary tree is the fundamental structure underlying a family of trees. These trees are used in many applications including search, sorting, compression, priority queues, and other data structures.

Note

Common binary tree data structures and applications:

- Heaps
- Treaps
- Binary tries
- Binary space partitioning
- Heapsort
- Priority queue
- Huffman compression
- Binary search tree

But the most common use of the binary tree is in the Binary Search Tree.

©2017 UMBC Training Centers

This page intentionally left blank.

CHAPTER 20

Binary Search Tree

Binary Search Tree (BST)

A **binary tree** with an ordering on node keys such that for any parent node, its left child has a lesser key and its right child has a greater key.

Thus from any node the left branch is followed given a lesser key and the right branch is followed for a greater key.

Then recursively for any node, its left subtree contains lesser keys and conversely its right subtree contains greater keys.

It follows that the minimum key is the smallest key in the left subtree and the maximum is the largest key in the right subtree, with respect to the root.

The keys can be returned in sorted order by a specifically ordered traversal.

BST property The left and right child keys are respectively less than and greater than their parent key.

©2017 UMBC Training Centers

Description

The binary search tree ordering permits fast search because entire subtrees can be skipped based on the outcome from comparing the search key with the key of the subtree root.

This is the primary use of a binary search tree.

The binary search tree is also used as a priority queue data structure because it is efficient to return the minimum or maximum key.

> The minimum is the leftmost node and the maximum is the rightmost node.
>
> The minimum or maximum may not be in a leaf node!

A binary search tree is given in Figure 20.1. Observe the ordering of values in the left and right subtrees of every node, and where the minimum and maximum numbers are located.

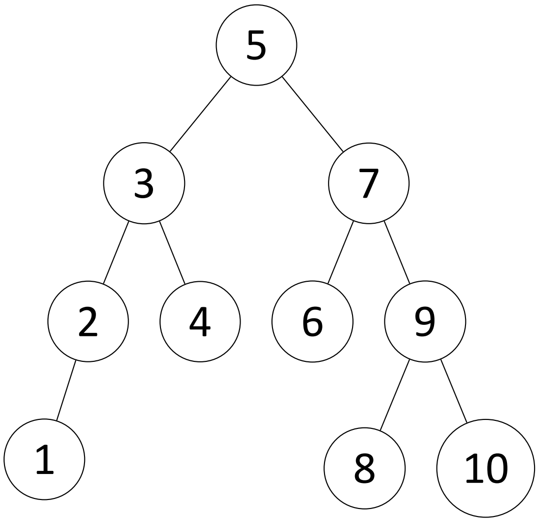


Figure 20.1: Binary Search Tree

The three primary operations on a binary search tree are search, insertion, and deletion.

> **Note**
>
> A binary search tree also supports finding the minimum or maximum.

Each of these operations must compare keys and preserve the BST property.

These operations each take $O(h)$ time where h is the height of the tree.

> The operations are optimal if $h = O(\log n)$.

It is natural to think of recursion for these operations because each branch is an independent result of a comparison.

Preliminaries

Every node has a key, a pointer for each child, and a pointer to the parent.

Thus each node is an object container that holds the following four items.

- key
- parent pointer
- left pointer
- right pointer

Let us first establish some preliminary notation.

We will use . and → as referencing operators on some object. Given an object n that has some member x, then we access that member using $n.x$ or $n \rightarrow x$.

These operators are interchangeable but we use → to give directional context, such as using node→left for pointing from a parent node to its left child.

Every node has an integer key.

A non-existing child is denoted by a null value.

Search

The search operation attempts to find some key in the binary search tree.

The first step determines if the key is equal to, less than, or greater than the root. The only operation in this first step is a comparison of the search key with the root key and therefore it takes $O(1)$ time.

The result of the key comparison decides whether the search should end or proceed down either the left or right branch.

The search continues down the tree from the root until either the key is found or a leaf node is reached, thereby ending the search.

Figure 20.2 depicts the search path for a key.

©2017 UMBC Training Centers

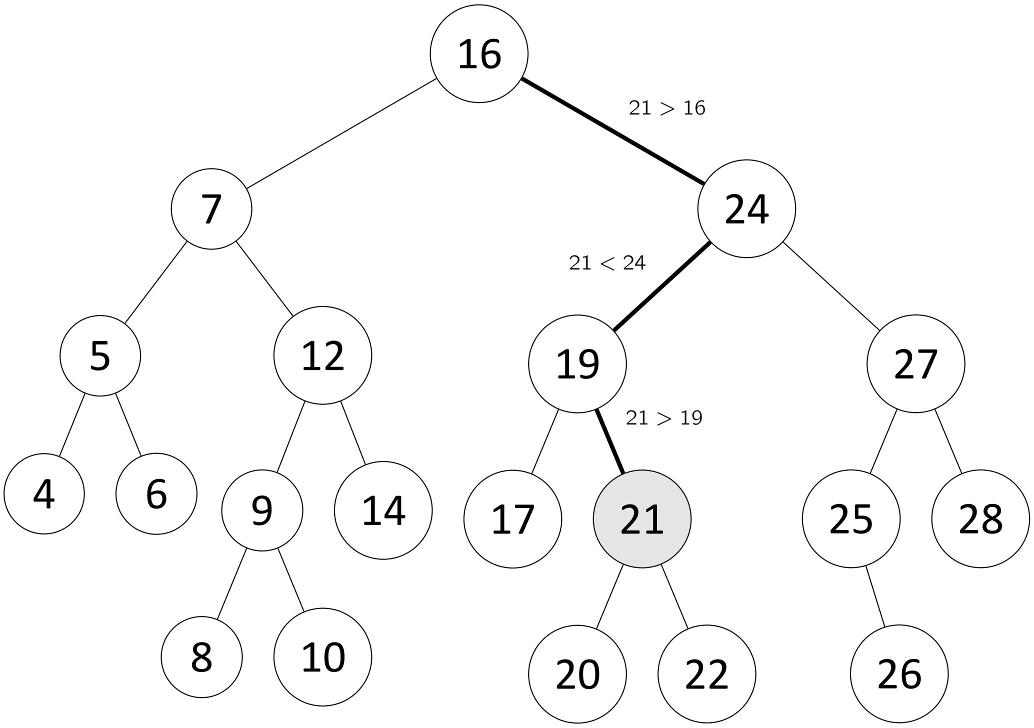


Figure 20.2: Searching for key 21 in a Binary Search Tree

©2017 UMBC Training Centers

A recursive algorithm for the search is immediately apparent.

```
Require: T                                              ▷ Binary Search Tree
Require: k                                              ▷ search key
  Set node := T$\rightarrow$root
  function SEARCH(node,k)
      if $k$ equals node.key or node.key equals $\emptyset$ then return node.key
      if $k <$ node.key then
          return SEARCH(node$\rightarrow$left, $k$)
      else
          return SEARCH(node$\rightarrow$right, $k$)
```

Observe that the search compares only a single node at each level of the tree. Moreover, once a branch is decided then the search will never proceed down the opposite subtree.

If the tree is complete then half of the remaining nodes are discarded from the search at each step. Therefore the work at each step is half the previous and the root level work is constant-time. This leads to the recurrence relation $T(n) = T(n/2) + 1$. Thus the search takes $O(\log n)$ time for a complete binary search tree (but not in general).

Note

A recurrence $T(n) = aT(\frac{n}{b}) + f(n)$, where $a, b > 0$, has $n^{\log_b a}$ leaf level work.

Here the leaf level work matches the root level work. The number of levels is $O(\log n)$ for a complete binary tree. The total work is 1 multiplied by the number of levels and therefore it is $O(\log n)$.

Searching is fast if the height of the tree is bounded by $\log n$, which is one of the key advantages of the binary search tree.

Insertion

A tree can be generated or built by adding one node at a time.

A null or empty tree has no nodes.

The first node added to an empty binary search tree is by default the root node.

The insertion of nodes to the tree must abide by the BST property.

An insertion of a new node follows the branching rules starting at the root and continues until an empty or open child position can be filled.

Thus each new node that is inserted starts out as a leaf node in the tree.

The insertion only requires finding a parent node that can accept the new node as a child, leading to just one child pointer update.

Figure 20.3 depicts the insertion of a new key.

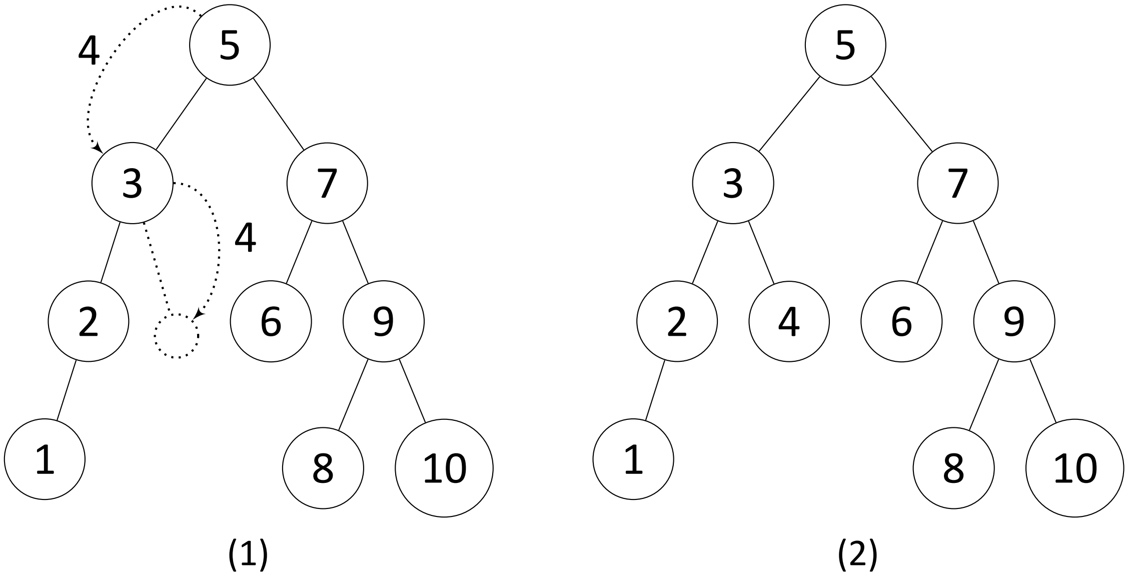


Figure 20.3: Inserting new key 4 in a binary search tree

The insertion operation finds a parent that can accept the new node as a child. In this sense the search operation is required.

Inserting a node is naturally recursive. The first step compares the new key with the root key. If the keys are equal then the new node has already been inserted. Otherwise either the left or right branch is followed depending on the sort order, and the process continues until completion.

The algorithm for insertion is similar to the search algorithm.

```
Require: T                                        ▷ Binary Search Tree
Require: i                                        ▷ new tree node
    Set node := T$\rightarrow$root
    function INSERT(node,i)
        if i.key equals node.key then return i
        if i.key $<$ node.key then
            if node$\rightarrow$left equals $\emptyset$ then
                set node$\rightarrow$left := i
                set i$\rightarrow$parent := node
            return INSERT(node$\rightarrow$left, i)
        else
            if node$\rightarrow$right equals $\emptyset$ then
                set node$\rightarrow$right := i
                set i$\rightarrow$parent := node
            return INSERT(node$\rightarrow$right, i)
```

At each level a single comparison is made with a node and the result decides which branch of the subtree rooted by that node will be fol-

©2017 UMBC Training Centers

lowed. In the best case this again leads to $O(\log n)$ time for insertion.

Note

A new node can also be inserted in-between a parent and child and still preserve the binary search tree ordering.

Deletion

Deletion is straightforward for a leaf or an internal node with one child.

An internal node with two children has only one parent, or no parent in the case of the root, thus removing such a node leaves two subtrees for just one available child slot, and the BST property must be preserved.

One method is to replace the deleted node with the minimum in its right subtree, and replace this minimum with its right child if one exists.

The following properties of a minimum node make the next operations possible.

- It cannot have a left child.
- It must be a left child.

Any key in the right subtree is greater than any key in the left subtree, so the right subtree's minimum can take the root of the left subtree as its left child.

The right subtree minimum can be replaced by its right subtree if one exists. Specifically, the minimum's right child becomes its parent's left child.

Figure 20.4 depicts the deletion of an internal node.

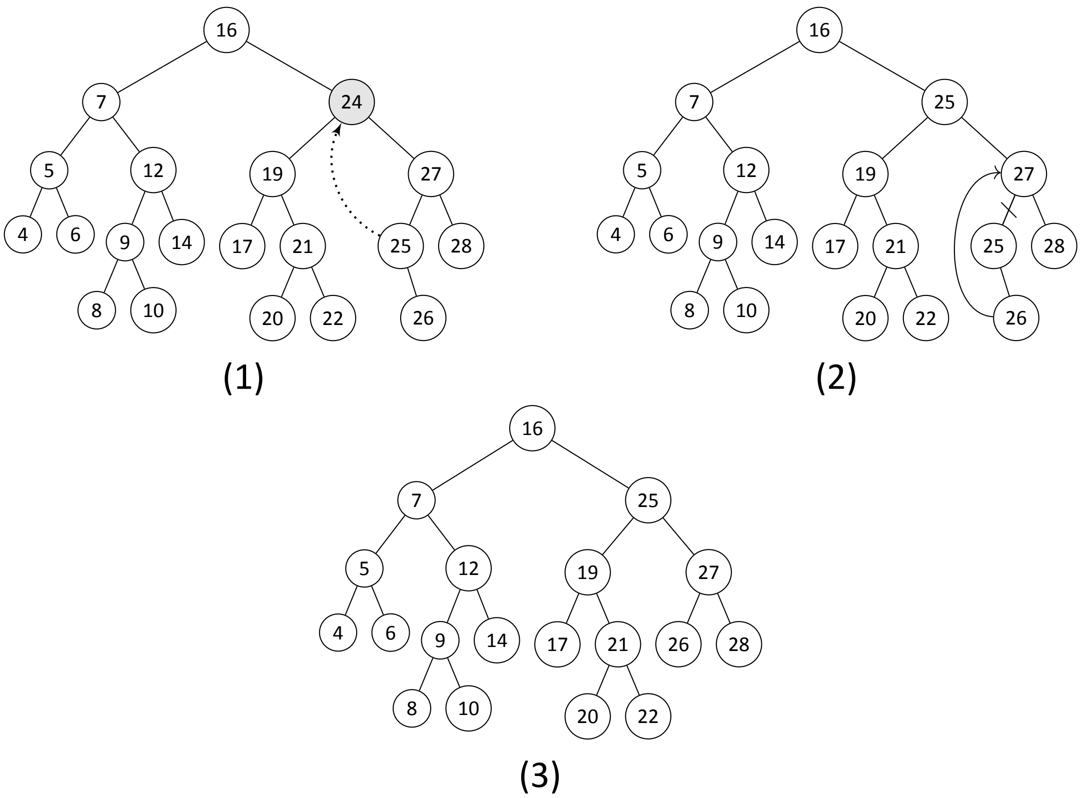


Figure 20.4: Deleting key 24 in a Binary Search Tree

The previously described deletion operations can be accomplished using these steps.

1. Overwrite deleted node's key with the minimum key in its right subtree.
2. Remove the deleted node's right subtree minimum, and if the minimum has a right child then make it the left child of the minimum's parent.

Deletion overwrites a key and replaces a left child by its right child.

Figure 20.5 gives an abstract depiction of node deletion.

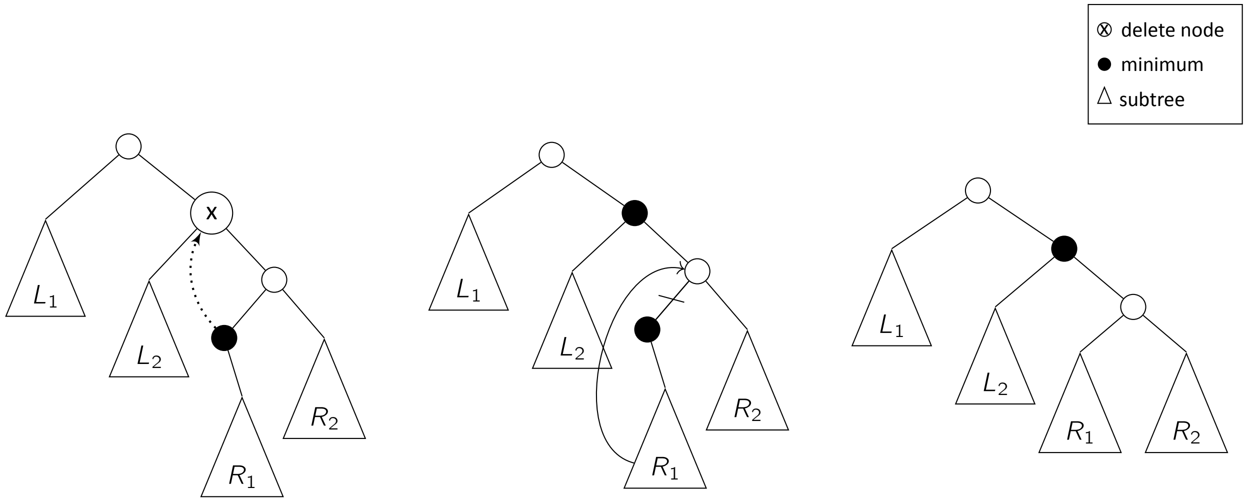


Figure 20.5: Deleting key x in a Binary Search Tree

Deletion algorithm

The algorithm for deletion follows; we leave handling of the root node as an exercise for the reader.

```
Require: T                                              ▷ Binary Search Tree
Require: x                                              ▷ node to be deleted
    Set root := T→root
    function DELETE(root,x)
        set node := SEARCH(root,x.key)
        if node.key equals $\emptyset$ then return
        if node has no right child then
            set parent's child pointer to node→left
            if node→left is not $\emptyset$ then
                set left child's parent pointer to node's parent
            delete node and return
        set min := minimum descendant in node's right subtree
        set node.key := min.key
        if node→right is min then
            set node→right := min→right
        else
            set min→parent→left := min→right
        delete min and return
```

Insertion Order

The binary search tree in Figure 20.6 can be generated by adding $5, 7, 3, 6, 4, 2, 9, 1, 10, 8$ sequentially while abiding by the ordering rule on left and right branching.

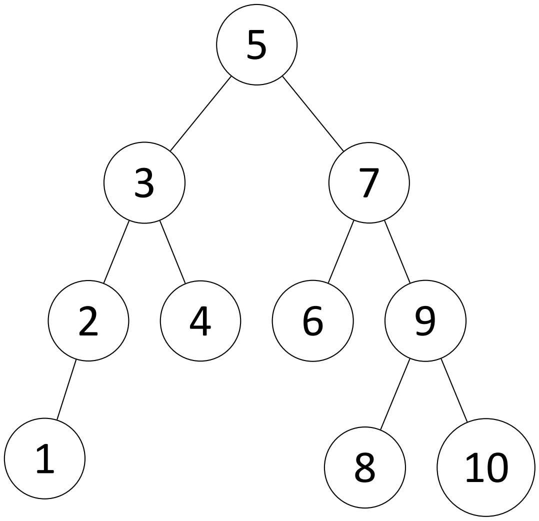


Figure 20.6: Binary Search Tree

It can also be generated by the sequence $5, 3, 4, 2, 1, 7, 6, 9, 8, 10$, which preserves the ordering of keys within the left and right subtrees.

But permuting the subsequence corresponding to the keys in a subtree will generate a different tree with the same root.

> Observe that the order of values in the subsequence corresponding to left and right child nodes does not matter because of the branching rule.

©2017 UMBC Training Centers

Suppose the $6, 7$ keys from the first sequence are exchanged.

$$5, 7, 3, 6, 4, 2, 9, 1, 10, 8 \longrightarrow 5, 6, 3, 7, 4, 2, 9, 1, 10, 8$$

This results in the new tree in Figure 20.7.

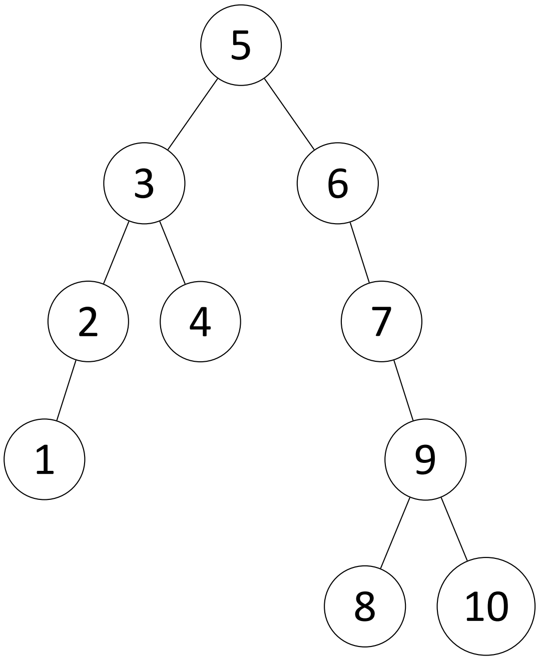


Figure 20.7: Binary Search Tree with $6, 7$ exchanged from the tree in Figure 20.6

The new tree in Figure 20.7 has a depth of four as opposed to a depth of three for the tree in Figure 20.1.

As discussed earlier, keeping the height of a tree at the lower-bound of $\Omega(\log n)$ is important for the performance of many applications.

What would the height be if the sequence were $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$?

It is possible to “balance” a tree and still preserve the BST property.

Exercise Set 6 - Trees

a) Understanding the structure of a tree is first and foremost when learning about tree data structures. Suppose you are given the pointer to a root of a binary tree that holds integers. Your job is to determine whether or not the tree is ”almost symmetric”.

That is, first, you will have to determine whether or not the tree is **structurally symmetric**. A tree is structurally symmetric if it has the exact same number of and exactly flipped structure of nodes on its left and right; its left and right subtrees must be exact **mirror images** of each other. Structural symmetry does not at all depend on the numeric values within those nodes.

Next, you will have to determine whether or not the tree is **numerically symmetric**. That is, you will need to determine the sum total of all the integers on the left subtree, and compare it to the sum total of all the integers on the right subtree. If they are equal, the tree is considered **numerically symmetric**.

In order to determine whether or not a tree is ”almost symmetric”, you will have to decide based on the two aforementioned symmetry criteria. A binary tree is **almost symmetric** if it is either structurally symmetric or numerically symmetric, **but not both**. In other words, it needs to have only one quality or the other in order to be considered ”almost symmetric”.

©2017 UMBC Training Centers

Write a function (you may approach this recursively or iteratively) that returns whether or not a binary tree is ”almost symmetric”, provided a pointer to the root node.

b) Given an array of integers size *n*, (given in any order) write a function that constructs an integer binary search tree from this array and returns a pointer to the root.

c) Write a function that, given a binary tree, determines the longest path from the root to a single leaf, and returns the size of such a path. Assume that none of the nodes maintain pointers to their parent nodes.

d) Write a function that, given a pointer to the root node of an integer binary search tree of size 2 or larger, returns the second smallest element.

This page intentionally left blank.

CHAPTER 21

Tree Traversal

Tree Traversal

A graph walk on a tree that visits each node once.

A graph walk proceeds from a node to an adjacent node until every node has been visited. This traverses the entire graph and is also known as a graph search.

Every node in a tree is processed in sequence starting from the root node by either breadth-first or depth-first search.

Different order of processing and recursive branching at each node generates different depth-first traversals.

Therefore the order in which nodes are processed depends on the traversal order.

Description

A tree can be traversed in a number of different ways by deciding which branch to take and when to process the node data. This leads to different traversal orders.

Here, processing the node data means that some function or operation is performed on the data. This could be simply reading or writing it out.

The sequence of processed nodes therefore depends on the traversal order, and traversal order depends on the sequence of actions taken at each node.

These actions include processing the node data and branching to its children.

This is fundamentally different than merely accessing the node data structure to get a reference to its parent or children.

Following pointers in a node-link data structure or calculating offsets in an array data structure is not considered processing the node data.

Hence a node that is *visited* during the tree traversal means that it was processed.

> Referencing a memory address of a node is not processing the node.

©2017 UMBC Training Centers

Since tree traversal processes every node only once, then traversing a tree of n nodes takes $O(n)$ time.

This is clearly evident for k-ary trees where the number of actions at each node in the traversal is fixed.

Every node in a graph, including a tree, can be found using breadth-first or depth-first search.

> **Note**
>
> A graph or tree can also be traversed using a random walk.

breadth-first search (BFS) traverses level-by-level. All nodes at the same level are visited before descending to the next level. Thus it broadens before deepening the search.

depth-first search (DFS) traverses branch-by-branch. All nodes in a path from start to leaf are visited before backtracking up one level and descending down the next path. Thus it deepens before broadening the search.

Tree traversal is naturally recursive because a tree is recursively defined.

Recursion on a binary tree can branch to either the left or right subtree. This leads to different sequential processing when traversing the tree.

The order in which nodes are processed depends on how recursion and processing are interleaved.

Next we'll describe each of these for binary trees.

Note

A typical BFS algorithm uses a queue (FIFO) and is therefore not strictly recursive.

A typical DFS algorithm uses a stack (LIFO) and is recursive.

Binary Tree Traversal

Binary Tree Traversal

A **tree traversal** on a binary tree.

By convention, depth-first traversal recurses down the left subtree before the right subtree of every node.

At each node in a depth-first traversal the following three actions are taken.

- Process current node
- Recurse left subtree of current node
- Recurse right subtree of current node

The depth-first traversal order is the order in which these action are taken.

The following lists the traversals under the main graph search method.

- depth-first search
  - pre-order
  - post-order
  - in-order
- breadth-first search
  - level-order

We'll begin with the depth-first search variants.

Note

There are other variants of the traversal orders including combinations of the above.

DFS Traversal

The depth-first search on a binary tree is simply a recursion on the left subtree followed by the right subtree for every node in a tree.

The depth-first traversal order determines the sequence in which nodes are processed.

Note

The convention is to recurse left then right, but recursing in opposite order is symmetrically equivalent.

Table 21.1 summarizes the recursion operations for the depth-first search traversal orders.

Table 21.1: Depth-First Traversal Order on Binary Trees

Pre-order	**Post-order**	**In-order**
process recurse left recurse right	recurse left recurse right process	recurse left process recurse right

©2017 UMBC Training Centers

Pre-order Traversal

Pre-order Traversal

A **tree traversal** on a binary tree that processes a node then recurses down the left subtree before recursing down the right subtree.

The order of operations at each node is then:

1. Process current node
2. Recurse left subtree of current node
3. Recurse right subtree of current node

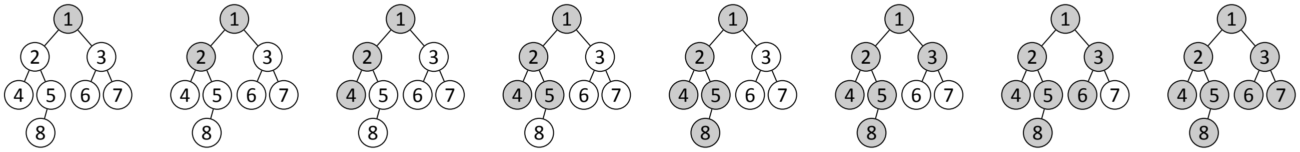
At each node in the recursion, process that node and try the left branch and on return, take the right branch.

A pre-order traversal is also known as *pre-fix order* traversal.

An algorithm that applies some function FN on all nodes in a binary tree using pre-order traversal is given by the following.

```
Require: T                                              ▷ Binary Tree
    Set node := T$\rightarrow$root
    function PREORDER(node,FN)
        if node equals $\emptyset$ then return
        FN(node)
        return PREORDER(node$\rightarrow$left, FN)
        return PREORDER(node$\rightarrow$right, FN)
```

Figure 21.1 illustrates a pre-order traversal on a binary tree.



Processing sequence: 1,2,4,5,8,3,6,7

Figure 21.1: Binary tree pre-order traversal.

Post-order Traversal

Post-order Traversal

A **tree traversal** binary tree that recurses down the left subtree of a node before recursing down the right subtree, then processes the node.

The order of operations at each node is then:

1. Recurse left subtree of current node
2. Recurse right subtree of current node
3. Process current node

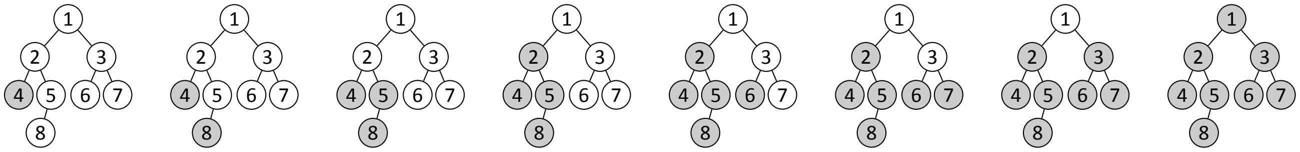
At each node in the recursion, try the left branch, and on return try the right branch, then on return process the node.

A post-order traversal is also known as *post-fix order* traversal.

An algorithm that applies some function FN on all nodes in a binary tree using post-order traversal is given by the following.

```
Require: T                                              ▷ Binary Tree
    Set node := T$\rightarrow$root
    function POSTORDER(node,FN)
        if node equals $\emptyset$ then return
        return POSTORDER(node$\rightarrow$left, FN)
        return POSTORDER(node$\rightarrow$right, FN)
        FN(node)
```

Figure 21.2 illustrates a post-order traversal on a binary tree.



Processing sequence: 4,8,5,2,6,7,3,1

Figure 21.2: Binary tree post-order traversal.

©2017 UMBC Training Centers

In-order Traversal

In-order Traversal

A **tree traversal** on a binary tree that recurses down the left subtree of a node then processes the node before recursing down the right subtree.

The order of operations at each node is then:

1. Recurse left subtree of current node
2. Process current node
3. Recurse right subtree of current node

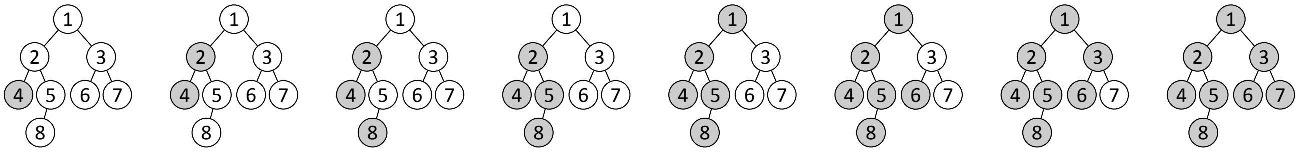
At each node in the recursion, try the left branch and on return process the node and then take the right branch.

An in-order traversal is also known as *in-fix order* traversal.

An algorithm that applies some function FN on all nodes in a binary tree using in-order traversal is given by the following.

```
Require: T                                              ▷ Binary Tree
    Set node := T$\rightarrow$root
function INORDER(node,FN)
    if node equals $\emptyset$ then return
    return INORDER(node$\rightarrow$left, FN)
    FN(node)
    return INORDER(node$\rightarrow$right, FN)
```

Figure 21.3 illustrates a in-order traversal on a binary tree.



Processing sequence: 4,2,8,5,1,6,3,7

Figure 21.3: Binary tree in-order traversal.

Iterative DFS

The previous recursive algorithms for depth-first search were based on a node-link abstract data structure of a binary tree.

Now recall that DFS has LIFO semantics and an algorithm can therefore employ a stack data structure.

> **Note**
>
> Recursive function calls are in fact placed onto a call stack, so in that sense the previously described algorithms are implicitly stack-based.

The traversal orders are equivalent whether an algorithm makes recursive function calls or implements the recursion using a stack data structure.

The following are iterative algorithms for the depth-first traversal orders.

Later we'll show that breadth-first search on a tree is inherently iterative.

The stack-based, iterative DFS traversal order algorithms are given next.

Iterative Pre-Order

An iterative pre-order traversal algorithm is given by the following.

Note

The right child is pushed onto the stack before the left child because of LIFO.

```
Require: T                                                    ▷ Binary Tree
Require: S                                                    ▷ stack
    set node := T$\rightarrow$root
    function PREORDER(node,FN)
        if node equals $\emptyset$ then return
        push node on S
        while S $\neq \emptyset$ do
            set node := pop from S
            FN(node)
            if node$\rightarrow$right $\neq \emptyset$ then push node$\rightarrow$right on S
            if node$\rightarrow$left $\neq \emptyset$ then push node$\rightarrow$left on S
```

Iterative Post-Order

Observe that a post-order traversal is the reverse of a pre-order traversal that recurses the right subtree before the left. This can be achieved easily using two stacks.

One stack holds the final sequence of nodes that need to be processed. The other stack is the working stack. On extraction from this working stack, the children of the extracted node are added to the stack and the extracted node is then added to the final processing stack.

Because of LIFO semantics, the left child precedes the right child in the working stack.

©2017 UMBC Training Centers

An iterative post-order traversal algorithm, using two stacks, is given next.

```
Require: T                                              ▷ Binary Tree
Require: P                               ▷ stack for final processing
Require: S               ▷ stack for children waiting to be added to P
    set node := T→root
    function POSTORDER(node,FN)
        if node equals $\emptyset$ then return
        push node on S
        while S $\neq \emptyset$ do
            set node := pop from S
            if node→left $\neq \emptyset$ then push node→left on S
            if node→right $\neq \emptyset$ then push node→right on S
            push node on P
        while P $\neq \emptyset$ do
            set node := pop from P
            FN(node)
```

A one-stack iterative post-order traversal algorithm is possible. We leave that as an exercise for the reader.

Iterative In-order

An iterative method for in-order puts all nodes on a left branch onto a stack until reaching a leaf. On extraction, the right child is put on the stack and the steps are repeated.

An iterative in-order traversal algorithm is given by the following.

```
Require: T                                             ▷ Binary Tree
Require: S                                             ▷ stack
    set node := T$\rightarrow$root
function INORDER(node,FN)
    while S $\neq \emptyset$ or node $\neq \emptyset$ do
        if node $\neq \emptyset$ then
            push node on S
            set node := node$\rightarrow$left
        else
            set node := pop from S
            FN(node)
            set node := node$\rightarrow$right
```

BFS Traversal

The breadth-first search on a binary tree explores each level completely before exploring the next.

In contrast the depth-first traversals recursively explore the left and right subtrees of each node. Since a subtree is itself a tree, the recursion is natural.

But the breadth-first traversal does not proceed by recursion over subtrees.

Instead some data structure is needed to line up nodes in each level.

Observe that FIFO semantics would achieve this result of processing every node in a level.

Therefore breadth-first search, known as level-order traversal, can employ a queue data structure.

> This is similar to LIFO semantics in non-recursive depth-first search algorithms where a stack was employed.

©2017 UMBC Training Centers

Level-order Traversal

Level-order Traversal

A **tree traversal** on a binary tree that processes every node in a level before descending to the next level.

It is a breadth-first search on the tree.

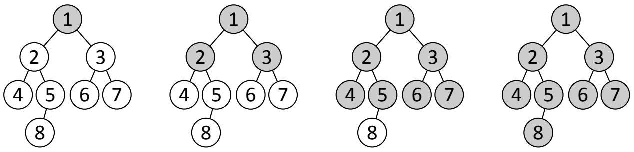
Level-order traversal on a tree processes every node level-by-level.

Since it is not a recursion, the order in which children of node are inserted into the queue does not matter.

A level-order algorithm that applies some function FN on all nodes in a binary tree

```
Require: T                                   ▷ Binary Tree
Require: Q                                   ▷ queue
  set node := T→root
  function LEVELORDER(node,FN)
      if node equals ∅ then return
      enqueue node into Q
      while Q ≠ ∅ do
          set node := dequeue from Q
          FN(node)
          enqueue node→left into Q
          enqueue node→right into Q
```

Figure 21.6 illustrates a level-order traversal on a binary tree.



Processing sequence: 1,2,3,4,5,6,7,8

Figure 21.4: Binary tree level-order traversal.

Binary Search Tree Traversal

The depth-first traversal orders have added meaning for binary search trees.

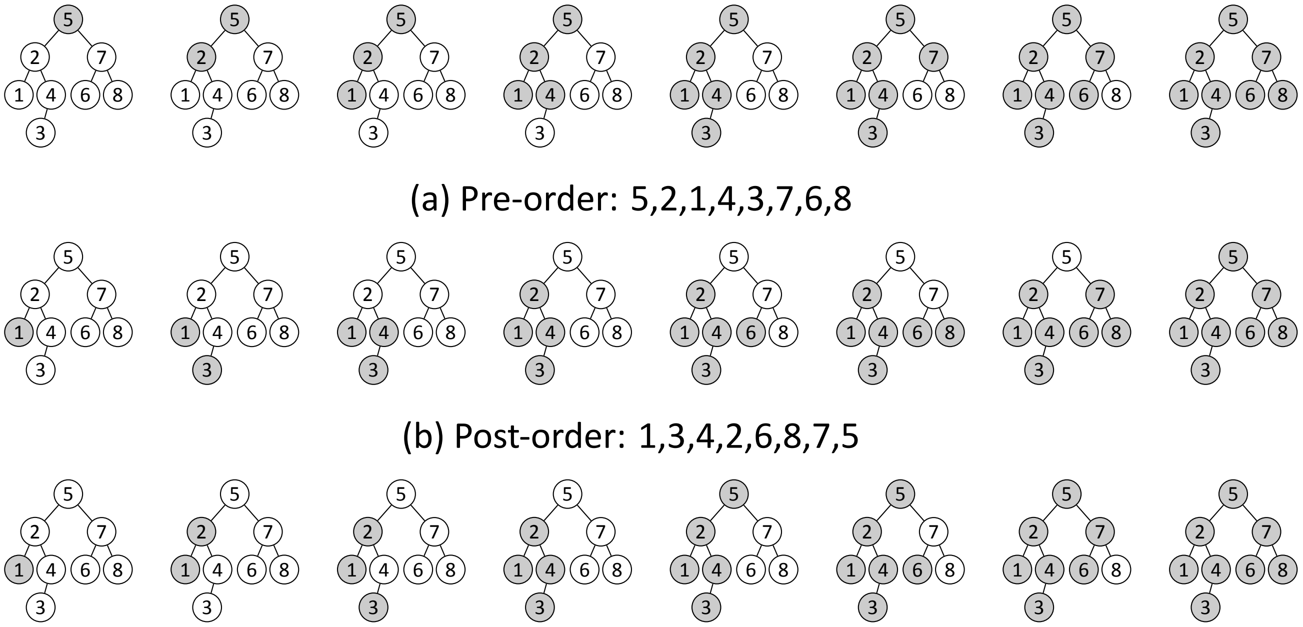
Nodes in a binary search tree follow a strict ordering imposed by the BST property. Specifically, all keys in the left subtree are less than the root key and all keys in the right subtree are greater.

It follows then that a depth-first ordered traversal can yield a sorted sequence of the nodes in the binary search tree.

The in-order traversal yields ascending sorted order.

> **Note**
>
> Recursing down the right subtree before the left subtree for in-order traversal gives nodes in descending sorted order.

A comparison of the traversal orders on a binary search tree is given in Figure 21.5.



(a) Pre-order: 5,2,1,4,3,7,6,8

(b) Post-order: 1,3,4,2,6,8,7,5

(c) In-order: 1,2,3,4,5,6,7,8

Figure 21.5: Binary search tree traversal.

©2017 UMBC Training Centers

Exercise Set 7: Tree Traversals

a) Tree traversals can get a little tricky- your job here will be to explore the use of a few types of traversals as you approach the **subtree sum** problem.

Given a pointer to the root of an integer binary tree, along with a specific integer contained in the tree, your job will be to determine the sum of all the integers within that element's subtree.

Here is an example. Suppose we are given the input as follows: a pointer to a root node (1), along with the number 2 as the start of our subtree sum. We would make use of a traversal starting at 2 in order to sum up all of the elements in 2's subtree, then return it. As such, 2's subtree sum would be $2+4+5+8=19$.

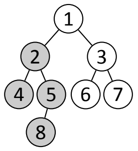


Figure 21.6: Subtree Sum example

Additionally, you are required to solve this problem **three ways**. You will write the function once making use of a Pre-order traversal, write it once more making use of a Post-order traversal, and write it a final time making use of an In-order traversal. You will write 3 functions:

– `subtree_sum_preorder`

– `subtree_sum_postorder`

– `subtree_sum_inorder`

b) Write a function that performs an **every-other** level-order traversal of a binary tree. That is, you are asked to write a function that, given a pointer to the root node of a binary tree, will print a level order traversal of all the **odd** levels in the tree. That is, you will print the root (level 1), then don't print level 2, print level 3, etc.

For example, if you were to perform the every-other level-order traversal of the tree detailed in the previous problem, you would return: `1,4,5,6,7`.

c) Given a pointer to the root of what is supposedly an integer binary search tree, write a function that determines whether or not it is a valid binary search tree. You are required to use a recursive approach.

d) Given a pointer to the root of an integer binary search tree, along with an integer representing a specific element, return the in-order predecessor of that element.

For example, if we were to use the given tree from part a, and asked to find the in-order predecessor of 2, your function would have to return 4. For full credit, do not perform a full in-order traversal each time this function is called.

This page intentionally left blank.

CHAPTER 22

Balanced Binary Search Tree

Balanced Binary Search Tree

A **binary search tree** in which the height of its subtrees are not very different.

The distance from node to leaf in a subtree is close for every node in the subtree.

Each subtree height is logarithmic in the number of nodes in that subtree.

Hence, a binary search tree is balanced if each subtree is itself balanced.

Description

The performance of search (and other operations) on a binary search tree is bounded by the tree height h, therefore taking $O(h)$ time.

At best it takes $\Omega(\log n)$ time and $O(n)$ time at worst.

The height ranges in the interval,

$$\Omega(\log_k n) \leq h \leq n.$$

Figure 22.1 illustrates the difference height can make for operations such as search.

©2017 UMBC Training Centers

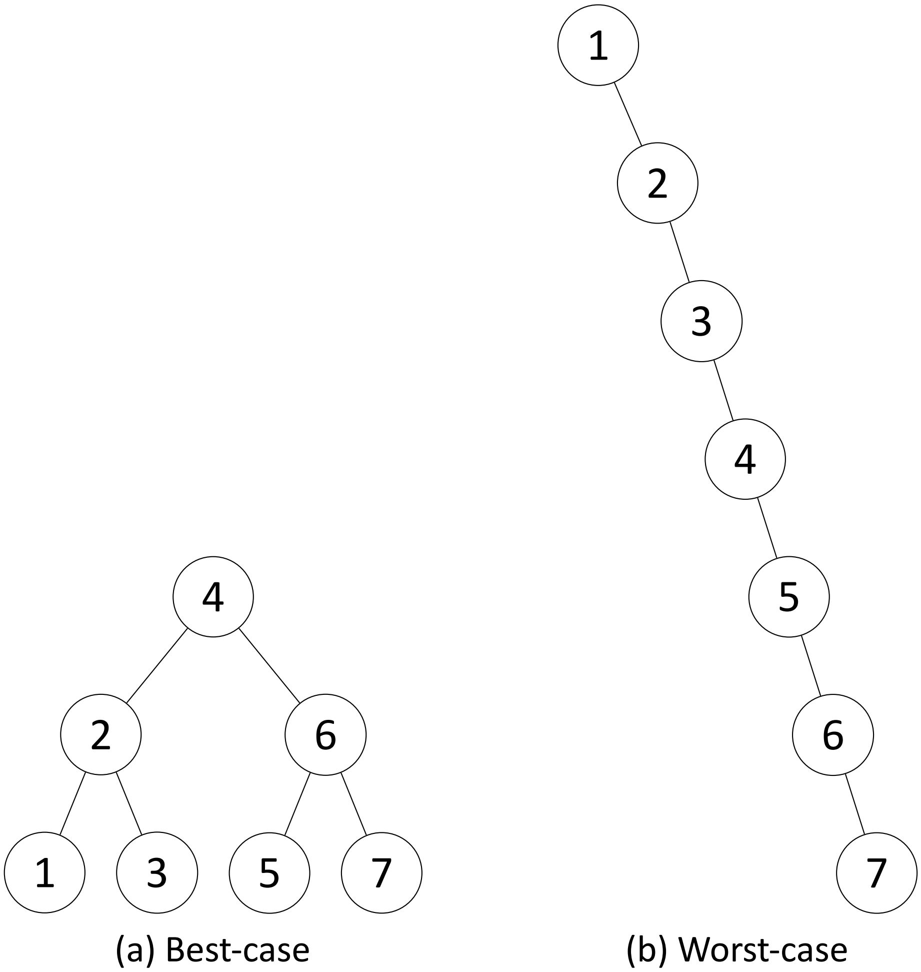


Figure 22.1: Best and worst tree shapes.

©2017 UMBC Training Centers

A perfectly “balanced” binary search tree has height on order of $\log n$ and therefore optimal for operations of insertion, deletion, and search.

> But perfect is hardly found in practice.

We do not have a precise definition of a balanced tree for any tree that isn’t perfect. But intuitively the distances from the root to every leaf should not be much different.

Recall that a subtree is itself a tree. Then a balanced tree is one in which its subtrees have height that is bounded by the logarithm of their nodes and hence each subtree is itself a balanced tree.

Thus we can recursively determine if a tree is balanced by this common sense notion.

Now let’s constrain the height difference to be at most one. This leads to the following new definitions.

node height The maximum distance from a node to a leaf descendant.

height-balanced node The subtrees of a node differ in height by at most one.

height-balanced tree Each node in a tree is height-balanced.

©2017 UMBC Training Centers

Note

This definition of height-balanced tree is used by the AVL tree.

Going forward we will use this more constrained definition for a height-balanced tree.

©2017 UMBC Training Centers

We compare the node heights to determine if a tree is balanced. A node is height-balanced if the height of its subtrees differ by no more than one. But an empty subtree must be included in this calculus.

- The root node has maximum height.
- An empty tree has height zero.
- Any leaf node has height zero.

Suppose a node has a right subtree but not a left subtree. Then the left subtree is an empty tree and hence its height is zero. This node is height-balanced only if the height of its right subtree is one.

Let’s compare the trees in the next figure.

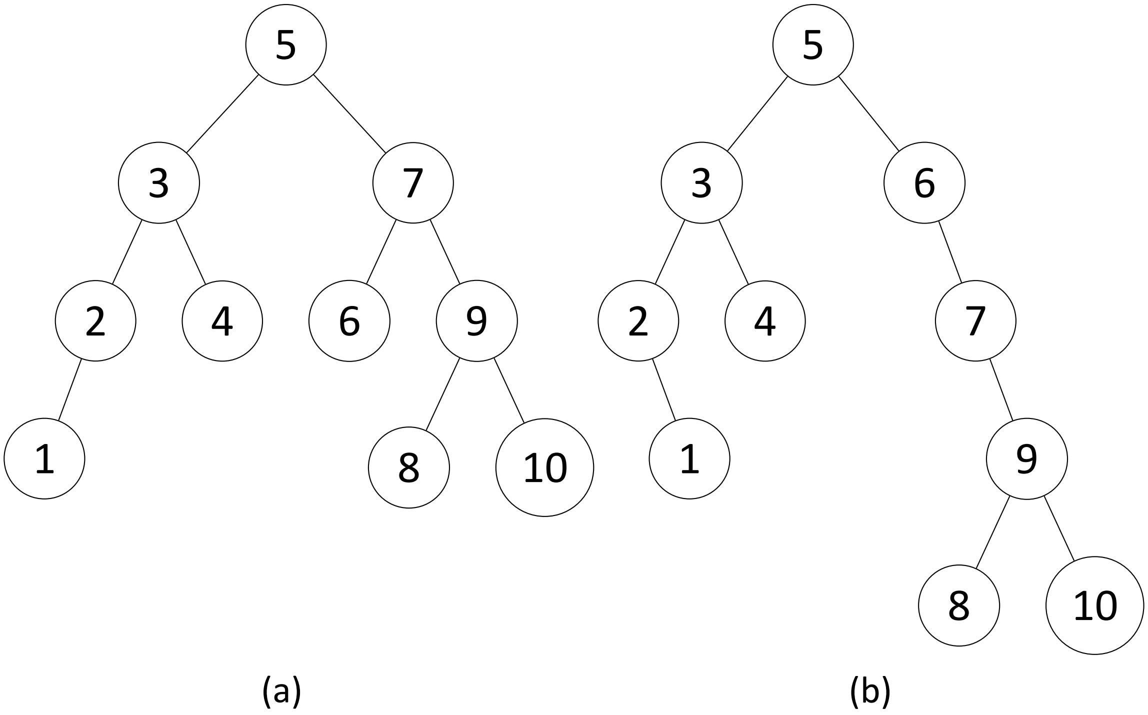


Figure 22.2: Height comparison

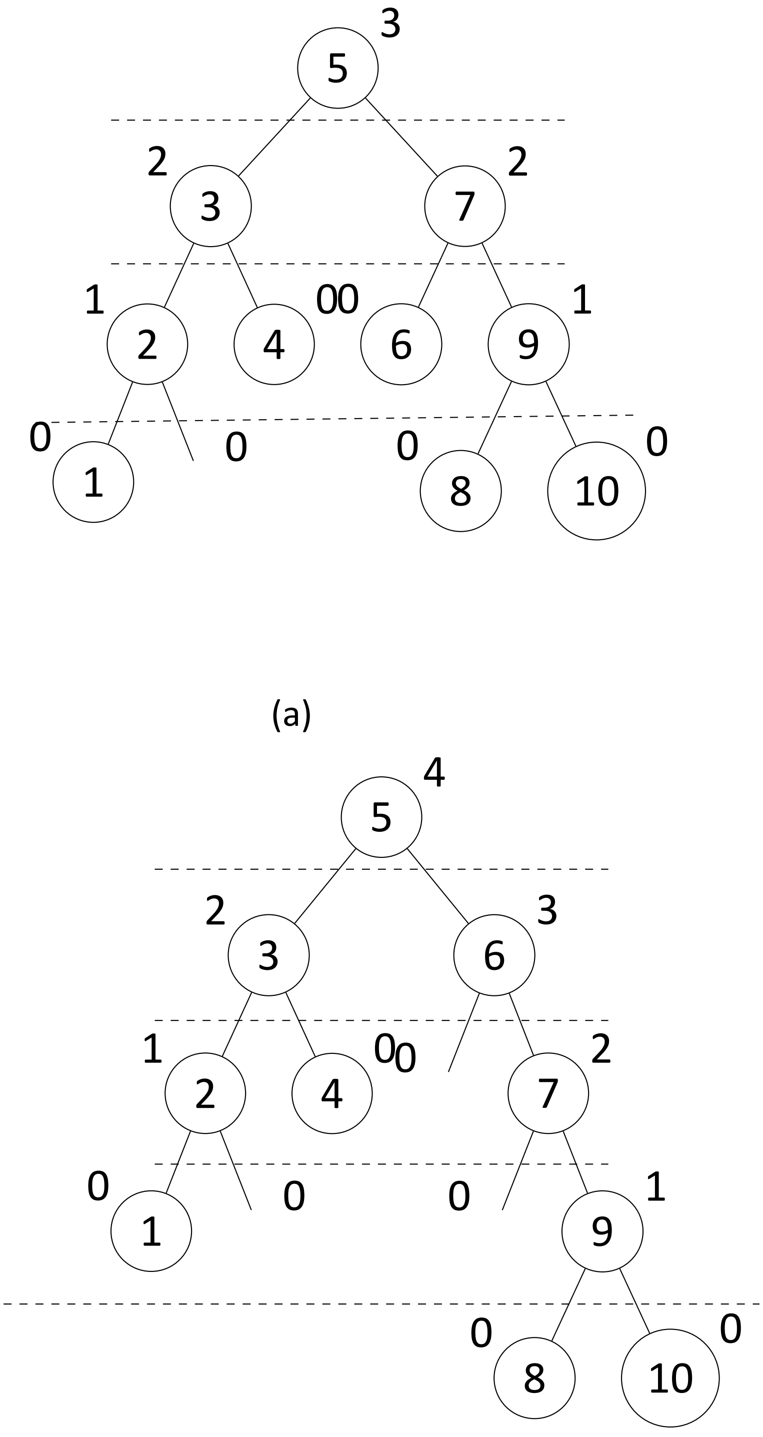
It should be clear that only the first tree in Figure 22.2 is balanced. Observe that node 6 in the second tree has no left subtree but its right subtree has height of two. Therefore the tree is unbalanced because the difference in the subtree heights is more than one.

Height imbalance is more apparent by decorating each node with its node height so then at each level the difference in node height of siblings can be directly compared. Hence, a tree is balanced if at every level, all siblings l, r with respective node heights h_l, h_r satisfy $|h_l - h_r| \leq 1$.

Note

A missing sibling is an empty tree, thus has height zero.

Decorating the trees in Figure 22.2 in this manner is illustrated in Figure 22.3



(a)

(b)

Figure 22.3: Decorated height comparison

It is readily apparent that the tree in Figure 22.3b is unbalanced because in level 2 the height difference between node 7 and its nonexistent sibling is two.

A balanced tree can become unbalanced as new nodes are inserted or deleted. This can be remedied by a sequence of “rotate” operations that exchange subtrees to keep the node heights with a difference of one.

Rotation

A node can be *rotated* to move it and one of its subtrees up one level while preserving the BST property. A rotation is therefore used to balance a tree. A node can be rotated right if it is a left child or left if it is a right child.

In a right rotation, the rotated node is a left child whose parent becomes its right child and its right child becomes its parent's left child.

A left rotation is just the opposite.

The right subtree of the rotated node becomes the left subtree of its parent, and in turn the node's new right subtree is rooted at its parent.

The net result is the rotated node and its left subtree move up one level, whereas the parent and the parent's right subtree move down one level. Therefore the left child and parent of the rotated node become siblings whose parent is the rotated node.

Only one subtree changes to the opposite side.

right rotation A right subtree becomes a left subtree.
left rotation A left subtree becomes a right subtree.

The operations for a right rotation on a node are summarized next and illustrated in Figure 22.4.

©2017 UMBC Training Centers

1. The node's parent becomes its right child.
2. The node's right child becomes the parent's left child.

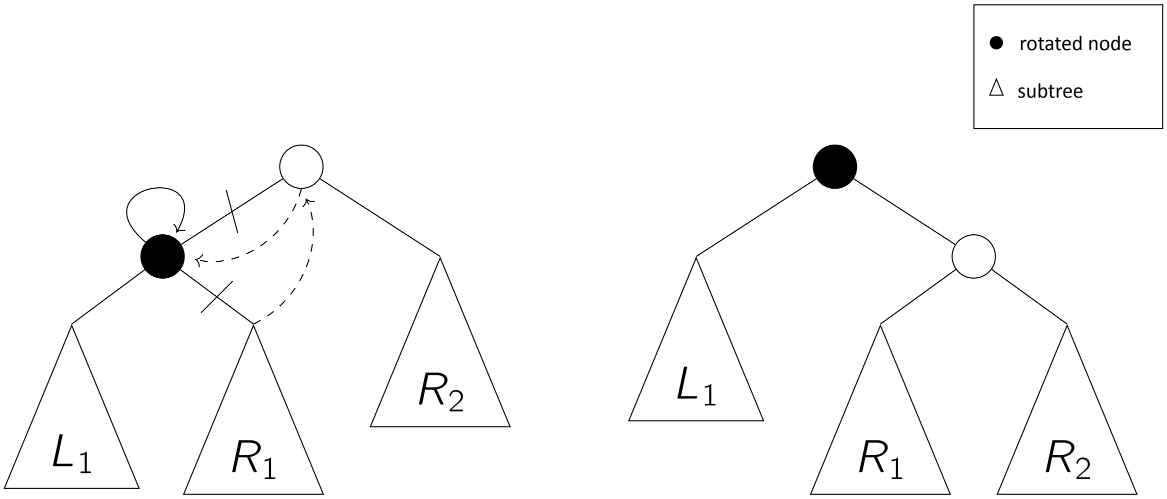


Figure 22.4: Right rotation of the solid node.

The rotation does not change the ordering on keys.

Before the rotation, all keys in the right subtrees of the rotated node and its parent are all greater than its key and any in its left subtree.

After the rotation, the parent gets the rotated node's right subtree and the rotated node's new right subtree is now rooted at this parent. Hence the ordering on keys is the same.

This is easily verified by looking at the only subtree that changes side.

In Figure 22.4, only the R_1 subtree changes side, going from a right-to-left subtree.

Observe that both before and after the rotation, all keys in R_1 are greater than the rotated node and less than the parent of the rotated node.

The other subtrees have not changed sides so their key ordering hasn't changed. Thus the new tree is still a binary search tree.

Rotation preserves the BST property.

©2017 UMBC Training Centers

The rotation operation is used to keep a tree height-balanced and maintain the BST property.

Let us return to our previous height comparison of the trees in Figure 22.3.

It should be clear that a left rotation on node 7 would balance the tree in Figure 22.3b.

Observe that a height-balanced tree could have also been achieved had the keys been inserted in a different order.

Choosing a random ordering on keys before insertion helps to avoid the worst-case height.

But as the tree grows, it becomes cost-prohibitive to randomize the keys and re-construct the tree.

In contrast, the cost of rotation is minimal, especially if only a few number of rotations are needed with respect to the number of other operations such as search, insertion, and deletion.

Three nodes are involved in a single rotation:

i) The rotated node,
ii) its parent,
iii) and its right or left child.

A single rotation takes constant-time because the maximum number of pointers that have to be changed is fixed and independent of the tree size or height.

How many pointers take new objects?

(Hint: A change to a child requires a change to its parent.)

Balancing a tree with a single rotation requires the least cost.

But often it takes a sequence of rotations to balance a tree. Take for example the unbalanced tree in Figure 22.5. It takes two rotations to balance it; a right followed by a left rotation.

©2017 UMBC Training Centers

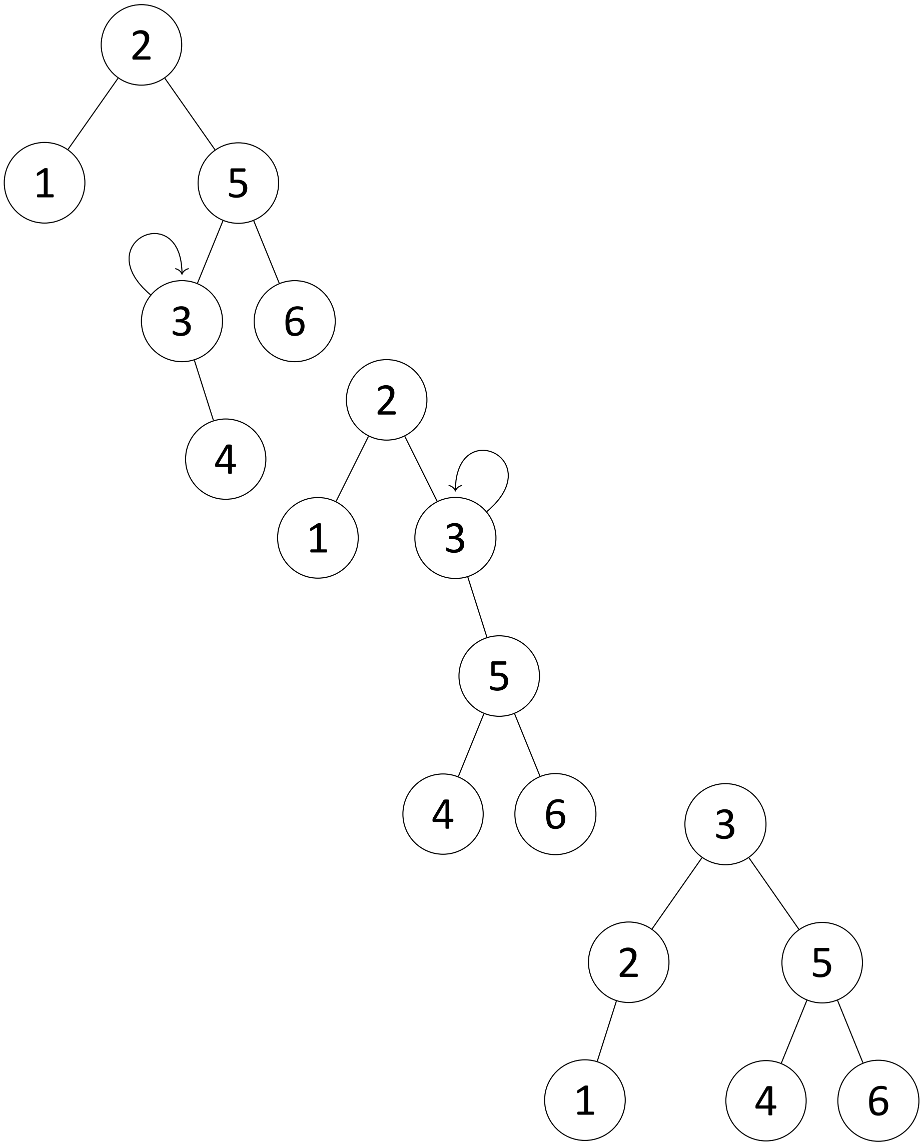


Figure 22.5: Double Rotation (Left to right: Right Rotation, then Left Rotation)

The algorithms for rotations are given next. We leave handling nulls and other details to the reader.

A right rotation changes the parent and right child of the rotated node.

```
function ROTATE_RIGHT(node)
    if node is $\emptyset$ or root or a right child then return
    set p := node→parent
    set pp := p→parent
    set c := node→right
    set node→parent := pp
    set node→right := p
    set p→parent := node
    set p→left := c
    set c→parent := p
    set pp’s appropriate child := node
    return node
```

A left rotation is the opposite of a right rotation.

A double rotation calls the single rotations in sequence.

```
function ROTATE_LEFT(node)
    set p := node→parent
    set pp := p→parent
    set c := node→left
    set node→parent := pp
    set node→left := p
    set p→parent := node
    set p→right := c
    set c→parent := p
    set pp’s appropriate child := node
    return node
```

```
function ROTATE_RIGHT_LEFT(node)
    return ROTATE_LEFT(ROTATE_RIGHT(node))
function ROTATE_LEFT_RIGHT(node)
    return ROTATE_RIGHT(ROTATE_LEFT(node))
```

Self-balancing

Keeping a binary search tree balanced ensures the height is $O(\log n)$ deep. It is necessary to maintain this bound on height to achieve optimal performance for the primary operations on the tree.

A tree can quickly become unbalanced. A single insertion or deletion can make a tree unbalanced.

Add node 11 to the balanced tree in Figure 22.3a.

It would be laboriously to manually check the tree after every modification. But with minor modifications to the tree data structure, a tree can rebalance itself.

self-balancing binary search tree
A **binary search tree** that automatically keeps each of its subtrees height-balanced.

A self-balancing binary search tree tracks the node heights after some arbitrary, but small, number of insertions or deletions. A sequence of rotations are made to rebalance the tree.

The obvious advantage is maintaining optimal runtime performance. But there are both space and time complexity costs needed to identify imbalance and correct it.

The costs include primarily the extra space needed to store the auxiliary information on subtree heights, and the time to evaluate these heights and make rotations to correct imbalance.

These costs are generally amortized over many operations to keep within optimal worst-case asymptotic bounds.

Recall there are $n = 2^{h+1} - 1$ nodes in a binary tree, implying $\log n$ lower-bound for the height.

A self-balancing binary search tree maintains the height to within a constant factor of $\log n$ to achieve the asymptotic lower-bound of $\Omega(\log n)$ depth.

The cost of re-balancing the tree is amortized over time as many more searches are performed than rotations.

Thus on average in the worst-case, the expensive operations can be kept close to the optimal runtime.

Self-balancing binary search trees underlie many important and common applications.

The first-known self-balancing binary search tree was the AVL tree.

Note

Other self-balancing binary search trees include:

- red-black tree
- splay tree
- treap
- AA tree
- scapegoat tree

©2017 UMBC Training Centers

AVL Tree

AVL Tree

A **self-balancing binary search tree** that is automatically height-balanced.

The subtrees of each node differ in height by at most one.

The subtree height for each node is maintained.

Rotations are automatically performed when subtrees differ in height by more than one.

Named after its inventors, Georgy M. Adelson-Velsky and Evgenii M. Landis.

AVL property The subtrees of each node differ by at most one.

Invented in 1962, the AVL tree was the first self-balancing binary search tree.

©2017 UMBC Training Centers

The AVL tree introduces the *balance factor*.

balance factor The difference in subtree heights for a node given by $h_R - h_L$, where h_R, h_L are the respective heights of the right and left subtrees of that node.

Note

Some authors use $h_L - h_R$ for the balance factor.

The AVL tree maintains the balance factor on each node and automatically performs rotations when subtrees differ in height by more than one.

A binary search tree is an AVL tree if every node has a balance factor in the range $-1, 0, 1$.

The worst-case height of an AVL tree is constant-factor of $\log n$, specifically it is $1.44 \log n$. This makes it very close to the minimum height.

Note

An AVL tree is a Fibonacci tree.

©2017 UMBC Training Centers

Before introducing AVL trees, we had considered a binary tree to be balanced if all sibling node heights h_l, h_r satisfy $|h_l - h_r| \leq 1$.

We defined the height of a node to be the maximum distance from that node to a leaf descendant, where both an empty tree and a leaf have zero height.

To align with the AVL tree balance factor arithmetic, we use the convention that an empty tree has height zero but a leaf has height one.

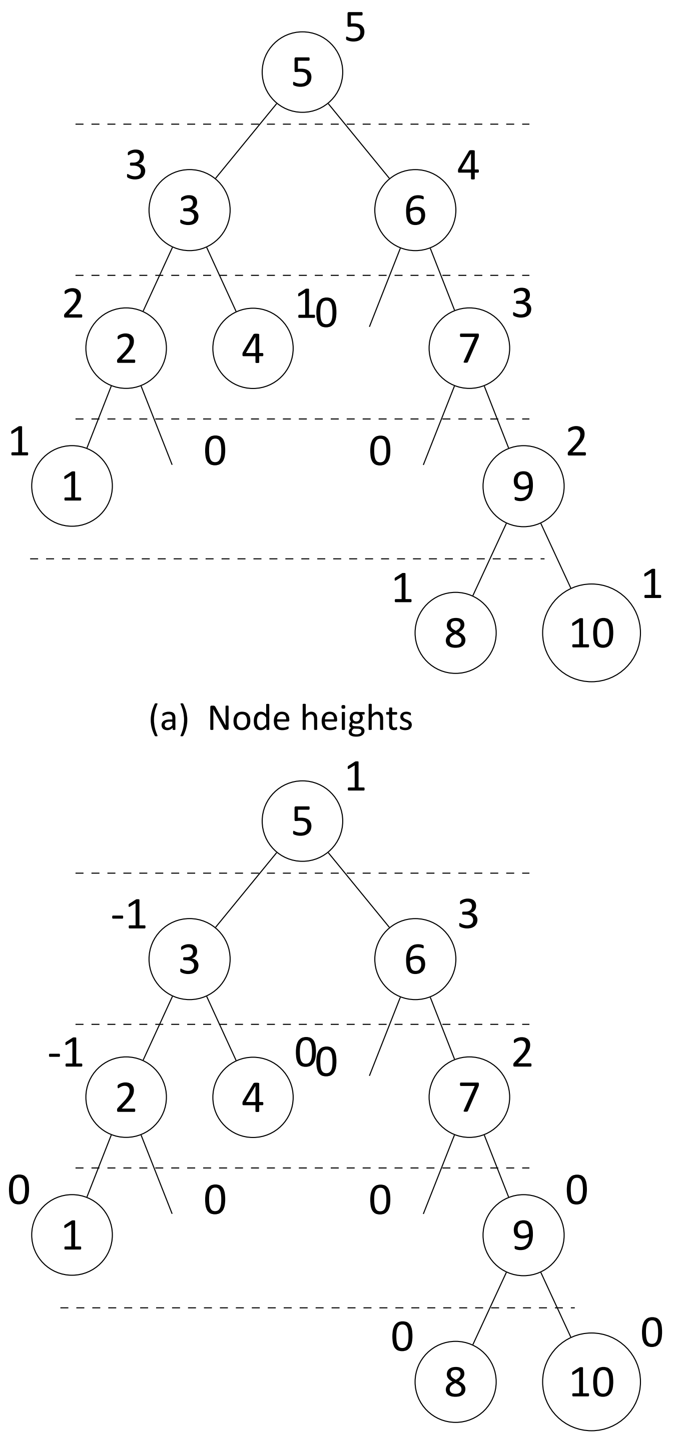
- An empty tree has height zero.
- Any leaf node has height one.

It still holds that the height of a node is one greater than the maximum height of its siblings.

Then for each node i with respective right and left child node heights, $h_R(i)$, $h_L(i)$, the balance factor bf for node i is given by,

$$bf(i) = h_R(i) - h_L(i).$$

Figure 22.6 illustrates this height convention and the corresponding balance factors.



(a) Node heights

(b) Balance factors

Figure 22.6: Node heights to balance factors

Maintaining the height for each node eliminates the expense of finding the longest descendant path for any node affected by a change to the tree.

When a change to the tree structure occurs the heights of the affected nodes must be adjusted, and if needed, rotations are used to rebalance the tree.

Let's begin with insertions.

Insertion

It is possible a tree remains an AVL tree after an insertion. Suppose the insertion violates the AVL property.

A newly inserted node begins as a leaf and only the nodes on the path from the root to this leaf are affected by the insertion, with each node in the path increasing in height by one.

Let x be the first in the path from the new leaf to root that is unbalanced. Then the difference in height between its subtrees, $|h_R - h_L|$, had to be exactly one prior to the insertion.

Let the heights of these subtrees be h, $h+1$ before the insertion.

The new leaf was added to the taller subtree so its height is now $h+2$.

It could not have been added to the shorter subtree. Why?

(Hint: Imbalance if $|h_R - h_L| > 1$.)

After the insertion, all nodes in the subtree rooted at x must have a balance factor of $|h_R - h_L| \leq 1$ because the tree was balanced before the insertion and the first violation of the AVL property is at x.

The insertion could have been in only four subtrees of x's grandchildren.

LL	x→left→left
LR	x→left→right
RR	x→right→right
RL	x→right→left

We will only cover the LL and LR cases since the RR and RL cases are exact mirrors.

Let's examine the first case.

In the LL case the new leaf is added to the subtree rooted at the LL grandchild of x.

The height of the LL grandchild is $h + 1$.

The heights of x$\rightarrow$left and x$\rightarrow$right are $h + 2$ and h, respectively, and therefore violate the AVL property.

Recall that in a right rotation, the left child and parent of the rotated node become siblings whose parent is the rotated node (see Figure 22.4).

Hence a single right rotation on x$\rightarrow$left makes the LL grandchild and x siblings.

After the rotation, x keeps its right child but its new left child was the right child of the LL grandchild. Therefore the height of x is now $h+1$.

The height of the LL grandchild remains at $h + 1$.

Now x and the LL grandchild are siblings and have the same height, therefore the tree is balanced.

Figure 22.7 illustrates the change.

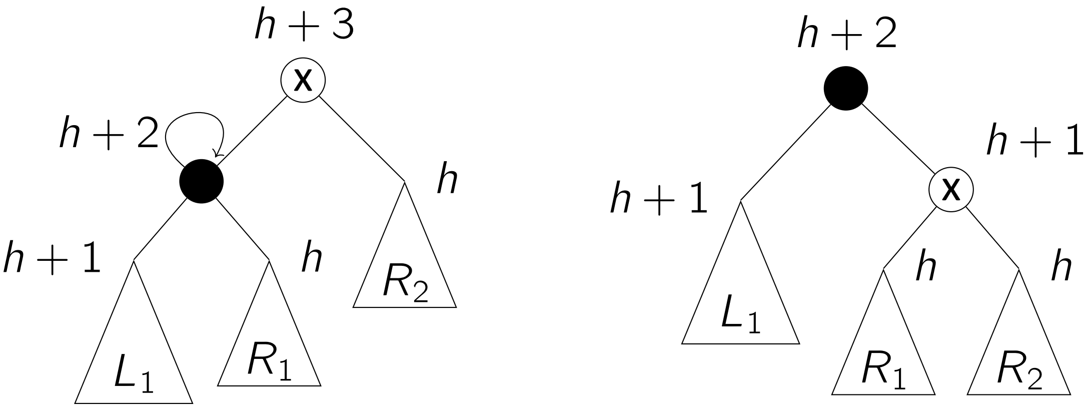


Figure 22.7: Rebalancing LL case.

Now consider the second case where the leaf is added to a subtree of the LR grandchild.

All the heights are the same as in the LL case, except the heights of the LL and LR grandchildren are opposite.

Since x is not height-balanced, then heights of x→left and x→right are $h + 2$ and h, respectively.

Then the LL and LR grandchildren have respective heights of h and $h + 1$.

This case requires a double rotation to re-balance the tree (see Figure 22.5).

We have asserted that x is the first unbalanced ancestor of the new leaf and the height of its LR grandchild before the insertion is h.

Therefore the subtrees of the LR grandchild had heights $h - 1$, $h - 1$, otherwise inserting into one of these subtrees would contradict our assertion.

Thus, after the insertion the subtree heights of the LR grandchild are h, $h-1$. We won't make a distinction between them in the next figure.

Figure 22.8 illustrates a left-right double rotation on the LR grandchild to rebalance the tree.

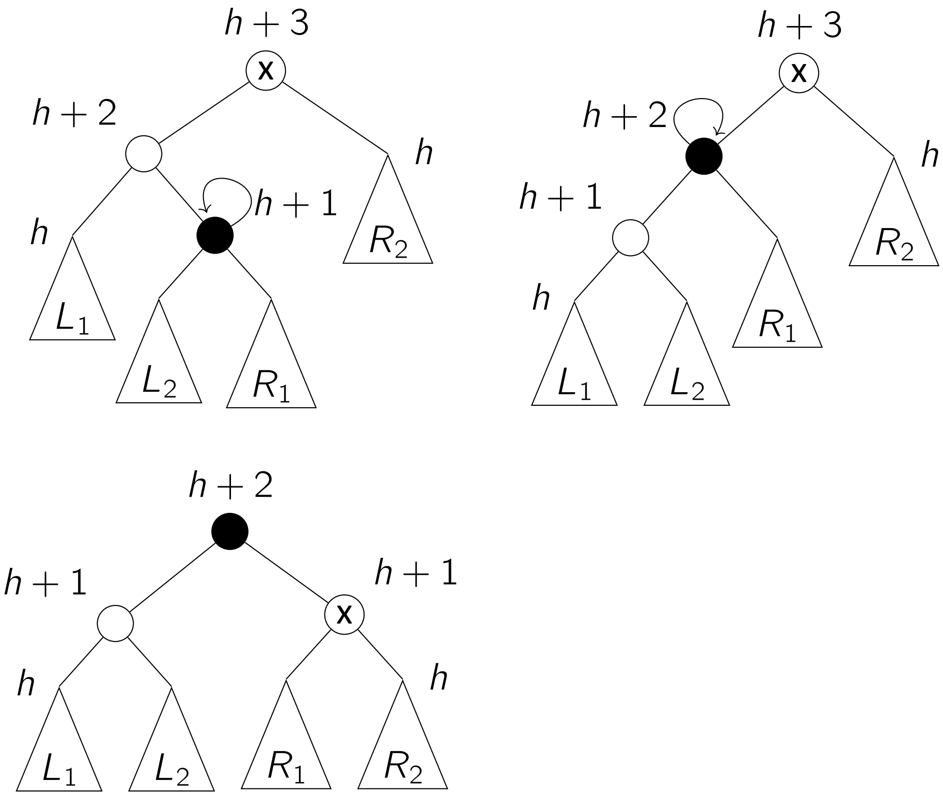


Figure 22.8: Rebalancing LR case.

The outer cases, LL and RR, require a single rotation on either the left or right child, respectively.

The inner cases, LR and RL, require a double rotation on the LR or RL grandchild, respectively.

LL	Single right rotation on left child.
RR	Single left rotation on right child.
LR	Double (L)eft-(R)ight rotation on LR grandchild.
RL	Double (R)ight-(L)eft rotation on RL grandchild.

After correcting the AVL violation, the insertion operation is complete and no further corrections are needed back up along the path from the new leaf to the root.

The cost of rebalancing after insertion is then $O(\log n)$.

A deletion operation also changes the structure of the tree and can cause a violation of the AVL property.

This can be corrected in a similar fashion as the insertion operation using rotations for the four cases.

But a deletion requires moving back up the tree to the root, correcting any AVL violations. Therefore rebalancing after deletion is more expensive.

Implementation

Every node in the AVL tree has a height data member in addition to the binary search tree data elements.

- key
- parent pointer
- left pointer
- right pointer
- height (or balance factor)

The height (or balance factor) is maintained accordingly on insertion and deletion operations.

On insertion, the node heights on the path from root to new leaf are incremented.

On finding an imbalance, which can only be one of the four cases, the appropriate rotations are performed and the heights for the nodes involved in the rotations are adjusted accordingly.

The rebalancing completes the insertion operation and no other traversals or height adjustments are needed.

The algorithm for an AVL tree insert follows. Null handling is left to the implementator.

```
Require: T                                                        ▷ AVL tree
Require: i                                                        ▷ new tree node
    Set node := T$\rightarrow$root
function AVL_INSERT(node,i)
    set node := INSERT(node,i)                                    ▷ BST insert
    node.height = 1;
    set $bf$ := 0
    while node$\rightarrow$parent $\neq \emptyset$ and $bf \leq 1$ do
        set node := node$\rightarrow$parent
        $bf \leftarrow |$ node$\rightarrow$right.height $-$ node$\rightarrow$left.height $|$
        set node.height := max(node$\rightarrow$right.height, node$\rightarrow$left.height) + 1
    if node$\rightarrow$parent equals $\emptyset$ or $bf \leq 1$ then return
    if i.key $<$ node.key then
        if i.key $<$ node$\rightarrow$left.key then
            set node := ROTATE_RIGHT(node$\rightarrow$left)
            set node$\rightarrow$right.height := node.height - 1
        else
            set node := ROTATE_LEFT_RIGHT(node$\rightarrow$left$\rightarrow$right)
            set node$\rightarrow$left.height := node.height
            set node$\rightarrow$right.height := node.height
            set node.height := node.height + 1
    else
        if i.key $>$ node$\rightarrow$right.key then
            set node := ROTATE_LEFT(node$\rightarrow$right)
            set node$\rightarrow$left.height := node.height - 1
        else
            set node := ROTATE_RIGHT_LEFT(node$\rightarrow$right$\rightarrow$left)
            setnode$\rightarrow$left.height := node.height
            set node$\rightarrow$right.height := node.height
            set node.height := node.height + 1
    return
```

The previous algorithm computes height differences. It isn't difficult to convert it to using balance factors, which has some advantages.

If re-balancing on every insertion or deletion, then balance factors have extrema values of $-2, +2$ and therefore requires only two bits of space. Moreover, the cases are easily deduced. Let x be the unbalanced node.

LL If balance factor is -2 and new key is less than x→left's key.

LR If balance factor is -2 and new key is greater than x→right's key.

RR If balance factor is $+2$ and key is greater than x→right's key.

RL If balance factor is $+2$ and new key is less than x→left's key.

The algorithm for deletion is similar to insertion. First the BST deletion operation is called. Any imbalance is corrected in the same manner as insertion, but the corrections must propagate along the entire path from the new leaf to the root. We'll leave it as an exercise for the reader.

Exercise Set 8: Balanced Trees

AVL Trees are excellent data structures that augment the binary search tree. They provide better guaranteed search times at the cost of a little more effort when you insert and delete. As you have observed in the chapter, an AVL tree will tolerate imbalance up to a certain level (balance factor).

a) Implement the function `is_balance_factor_satisfactory`, which takes a pointer to a node in a binary tree, along with a positive integer k, and then decides if the absolute value of the balance factor is less than or equal to k. You are essentially asked to write a function that determines whether or not the balance factor of a node is satisfactory, based on given conditions.

b) Using the function you produced in part a, implement an AVL-5 tree, which functions the exact same as an AVL tree, except it allows for the absolute value of the balance factor at every node to be less than or equal to 5. You are encouraged to let the function pseudocode in the text guide your implementation.

©2017 UMBC Training Centers

This page intentionally left blank.

Chapter 23

Huffman Code

Huffman code

A binary **prefix-free code** for data compression.

The Huffman code is an optimal prefix-free code that ensures the most efficient binary encoding for a string symbols.

It maps a sequence of bits to each symbol such that the most frequent symbols get the fewest bits.

This is accomplished by producing a rooted binary tree with symbols at the leaves and opposite branches get a 0 or 1 bit, respectively.

The path from root to symbol gives the codeword for that symbol, where the most frequent symbols are closest to the root.

The code is prefix-free, meaning no symbol can be the prefix of another and hence there is no ambiguity in decoding.

The Huffman algorithm generates the prefix-free tree in $O(n \log n)$ time.

It is a greedy algorithm that recursively merges the two least frequent symbols into a new symbol whose frequency is the sum of the two symbols.

Huffman coding requires the frequency of the symbols before the coding process begins. Thus it depends on the statistical properties of the input.

The *Huffman prefix code* was invented by David A. Huffman in 1951 and later published in 1952.

Description

The Huffman code is derived from a rooted binary tree where leaves are symbols and opposite branches get 0 and 1 bits, respectively. The path from root to leaf gives the codeword for the symbol denoted by the leaf.

Since each symbol is leaf, then it is not possible to get a codeword that is the prefix of another.

The most frequent symbols are closest to the root and therefore have the shortest codeword.

The tree is generated by the Huffman greedy algorithm.

The algorithm takes the two least frequent symbols and combines them into a new symbol with frequency being the sum of the two merged symbols, and recurses.

The algorithm is greedy because the greedy choice is to pick the two least frequent symbols at each step.

Note

As a doctoral student at MIT in 1951, David Huffman took a course on Information Theory by Robert Fano, who gave the students the option to either complete a term paper on an optimal prefix-free code or take the final exam. Huffman chose the term paper. Nearing the deadline and seemingly unsuccessful, Huffman resigned to take the final exam and began to discard his paper when the solution struck him like lightning.

The renowned Robert Fano had worked with Claude Shannon who was of even greater renown for starting the field of Information Theory. Both Fano and Shannon had struggled but were unsuccessful in finding an optimal prefix-free code.

Prefix code

A binary prefix code maps a string with a fixed alphabet to a bit sequence.

It is a sequence of codewords, each of which maps a single symbol in the string.

A prefix code is "prefix-free" meaning each codeword cannot be a prefix of another.

– Avoids ambiguity leading to unique decoding.

Here is an example that is not prefix-free.

> If "a" maps to 1 and "b" maps to 11, then the code 1111 decodes to "aaaa", "bb", "aab", "baa".

Consider an alphabet $A = \{a, b, c, d\}$ and the following binary prefix code that uses 2 bits to encode each character.

$$\begin{aligned} a &\mapsto 00 \\ b &\mapsto 01 \\ c &\mapsto 10 \\ d &\mapsto 11 \end{aligned}$$

©2017 UMBC Training Centers

Given a string S of 1000 symbols from alphabet A, then this encoding takes 2000 bits. But suppose “a” is 50% of the string, “c” is 25%, and “b” and “d” are each 12.5% of the symbols. Now use the next mapping.

$$\begin{aligned} a &\mapsto 0 \\ b &\mapsto 110 \\ c &\mapsto 10 \\ d &\mapsto 111 \end{aligned}$$

The string S is encoded in,

$$.5(1) + .25(2) + .125(3) + .125(3) = 1.75 \text{ bits per symbol.}$$

This saves 12.5% in bits ($\frac{2-1.75}{2} = \frac{.25}{2} = .125$).

Prefix-free code

Compression is due to using fewer bits for the most frequent symbol.

The code can be illustrated by a binary tree. Each symbol is a leaf so it cannot be the ancestor, and thus a prefix of another symbol.

Bits “0” and “1” correspond to the left and right branches, respectively.

> What happens if instead 1 and 0 were for the left and right branches, respectively?

Then a path from root to leaf gives the codeword for the symbol assigned to that leaf; appending bits to the right.

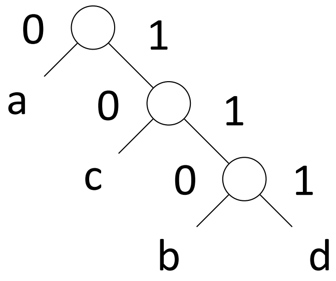
The length of the path is the length of the codeword for that symbol.

In an optimal encoding the most frequent symbols are closest to the root.

Then the most frequent symbol gets the shortest code and the least frequent gets the longest.

Figure 23.1 illustrates an example Huffman code tree and the derived codewords.

©2017 UMBC Training Centers



$$a \mapsto 0$$
$$b \mapsto 110$$
$$c \mapsto 10$$
$$d \mapsto 111$$

Figure 23.1: Prefix-free code tree.

©2017 UMBC Training Centers

Optimal Prefix-free code

It is challenging to find an optimal binary prefix-free code (prefix-code).

Such a encoding must ensure the most efficient mapping of symbols from any alphabet for any length string.

A rooted binary tree solves the prefix-free problem. But designing a method to place symbols in the tree that always results in an optimal code is non-obvious.

The insight from Huffman was to build the tree “bottom-up”.

The problem statement for an optimal binary prefix-free code is given next.

> Problem: Given an alphabet and frequency for each symbol, find a binary prefix code that gives an optimal encoding, meaning the shortest possible code.
>
> Minimize the total encoded length, $\sum_{i=1}^{n} p(i)d(i)$, where $p(i)$ is the probability and $d(i)$ is the depth of symbol i.

A Huffman code is such an optimal code.

©2017 UMBC Training Centers

Huffman Code

The Huffman algorithm builds a binary prefix tree in “bottom up” fashion, merging the bottom leaves first.

“Merge the two least frequent letters and recurse”

Merging two leaves creates a new internal node that is treated as a new symbol with frequency being the sum of frequencies of the merged nodes.

Merging ends when no other symbols can be combined, leaving a final optimal prefix code tree.

©2017 UMBC Training Centers

Illustration

Figure 23.2 illustrates the generation of the Huffman tree and final codeword mapping.

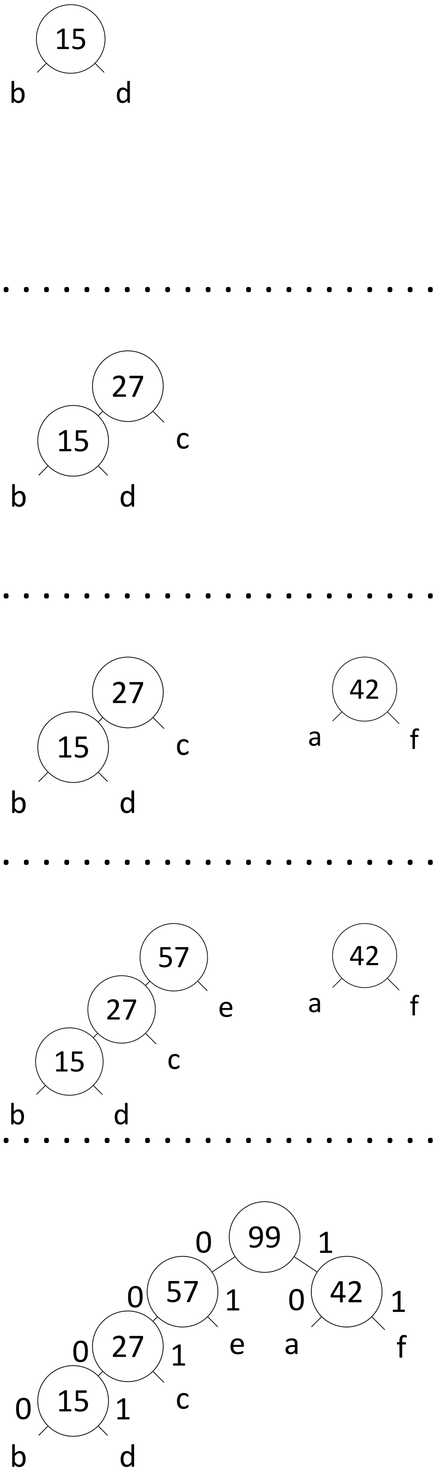
symbol	frequency
a	20
b	10
c	12
d	5
e	30
f	22

symbol	frequency
a	20
bd	15
c	12
e	30
f	22

symbol	frequency
a	20
bdc	27
e	30
f	22

symbol	frequency
af	42
bdc	27
e	30

symbol	frequency
afe	42
bdce	57



Final Encoding

symbol	codeword
a	10
b	0000
c	001
d	0001
e	01
f	11

Figure 23.2: Huffman Binary Prefix-free code tree.

©2017 UMBC Training Centers

Huffman Algorithm

Let Q be a priority queue (min heap)
Let A be an alphabet of n symbols
Let F hold the frequency/probability of each symbol in A
Let L,R,P be arrays to store the prefix code tree where L stores the left child, R stores the right child, and P stores the parent.

```
Require: alphabet $A$ of $n$ symbols
Require: array $F$ with frequencies for each symbol
Require: arrays $L$, $R$, $P$ to store prefix code tree
Require: priority queue $Q$ (min heap) sorted by $F$
function HUFFMAN(A,F)
    $n = |A|$
    for $i \in A$ do
        insert $i$ into $Q$
    for $i = 1$ to $n - 1$ do
        $z \leftarrow$ new internal node
        $x \leftarrow$ extract from $Q$
        $y \leftarrow$ extract from $Q$
        $F[z] \leftarrow F[x] + F[y]$
        $P[z] \leftarrow z$
        $L[z] \leftarrow x$
        $R[z] \leftarrow y$
        insert $z$ into $Q$
    return extract from $Q$
```

Initializing Q takes the time to build a min heap, thus $O(n)$ time.

©2017 UMBC Training Centers

Each queue operation takes $O(\log n)$ time because of heap property.

There are at most $n - 1$ iterations and therefore that many queue operations. Thus the algorithm takes $O(n+(n-1)\log n) = O(n\log n)$ time.

> **Note**
>
> The implementation of the priority queue in the Huffman algorithm is not limited to the heap data structure. Other possible methods include:
>
> - self-balancing binary search tree
> - sorted linked list with bisection search to maintain sort order

©2017 UMBC Training Centers

Optimality

The Huffman algorithm generates a binary prefix-free code tree from which the codewords for each symbol can be acquired.

The encoding of symbols from this result is provably optimal, meaning it encodes a string with the minimum number of bits.

The proof of the optimality is beyond the scope of this discussion, but we give the basic sketches in the next sections for the interested reader.

Huffman Optimality - Part I

Claim 1. *Let x, y be the least frequent symbols, then there is an optimal binary prefix code tree in which x, y are siblings at the maximum depth.*

Proof. Suppose T is an optimal binary prefix code tree containing siblings x, y but siblings b, c are at the maximum depth of T. Let $p(i)$ be the probability of symbol i and $d(i)$ is the depth of i in T. Since x, y are the two least frequent symbols then $p(x), p(y)$ are both less than $p(b)$ or $p(c)$, but $d(b), d(c)$ are both greater than either $d(x)$ or $d(y)$.

Now exchange x with b to get a new tree T'. The difference in cost between T' and T is due only to the costs of x and b in the two trees. Let $B(T) = \sum_i p(i)d(i)$ be the cost of a tree, thus

$$\begin{aligned}
B(T') - B(T) &= (p(x)d(b) + p(b)d(x)) - (p(x)d(x) - p(b)d(b)) \\
B(T') &= B(T) - d(b)(p(b) - p(x)) + d(x)(p(b) - p(x)) \\
&= B(T) - (p(b) - p(x))(d(b) - d(x)) \\
&\leq B(T). \\
&\qquad \text{(since } p(b) - p(x) \geq 0, d(b) - d(x) \geq 0\text{)}
\end{aligned}$$

The cost of T' is at most that of T and since T was optimal then T' is optimal. Similarly, exchanging y and c must give a new optimal binary prefix code tree, where this final tree has x, y at the maximum depth.

□

Huff Optimality - Part II

Claim 2. *Let T_n be an optimal prefix code tree satisfying Claim1 for n symbols. Let T_{n-1} be the tree of $n-1$ symbols after merging the siblings x, y into a new leaf node z having probability $p(z) = p(x)+p(y)$. The cost of a tree is $B(T) = \sum_i p(i)d(i)$ then $B(T_{n-1}) = B(T_n) - p(z)$. Thus the cost of T_{n-1} is less than that of T_n.*

Proof. Let d denote the depth of x, y in T_n, then z is at depth $d-1$ in T_{n-1}. Thus,

$$\begin{aligned}
B(T_n) - B(T_{n-1}) &= d(p(x) + p(y)) - (d-1)p(z) \\
&= dp(z) - (d-1)p(z) \\
&= p(z) \\
B(T_{n-1}) &= B(T_n) - p(z)
\end{aligned}$$

□

Huffman Optimality - Part III

Lemma 1. *The Huffman algorithm produces an optimal prefix code tree.*

Proof. We prove this by induction on the number of symbols, n. The base case $n = 1$ follows trivially. For $n \geq 2$, Claim 1 establishes that the two least frequent symbols, x and y, are siblings at the maximum depth of an optimal prefix code tree, T_n.

The algorithm merges x, y into a new symbol z whose frequency $p(z)$ is the sum of the x, y frequencies, producing a T_{n-1} tree with $n-1$ symbols. This new tree is lower in cost than T_n by amount $p(z)$ according to Claim 2. Since T_n is optimal then T_{n-1} is optimal.

By induction over the remaining $n-1$ symbols, the algorithm produces a final optimal prefix code tree. □

Chapter 24

LZ Compression

LZ coding

A **dictionary** coding method for data compression.

The LZ coding method is a family of dictionary coders.

These coders do not require knowledge of the statistical distribution of symbols in the input.

Instead a dictionary of codewords is generated during the coding process. A substring is encoded by matching it in the dictionary.

The LZ coding results in lossless compression.

The *LZ* compression technique was invented by Abraham Lempel and Jacob Ziv in 1977, and is now known as LZ77 coding.

©2017 UMBC Training Centers

Description

The original LZ coding method (LZ77) uses a sliding window, which is an implicit dictionary, to search for previously stored patterns.

A dictionary coder like LZ maintains a data structure, the dictionary, to store the substrings to be matched in the input.

On a match, a reference or index to the substring in the dictionary is substituted in the compressed output.

Thus coded substrings are referenced by their index in the dictionary.

Since the dictionary references are shorter in length, it leads to compression.

The LZ77 coder uses a sliding window for the dictionary. This window holds the last N processed bytes and as the window slides it encodes each new substring.

A 3-tuple is used to reference matched substrings found by the sliding window.

- distance from the current position (cursor) back to the start of a matching substring
- length of matching substring including any characters forward of the current cursor within some readahead buffer

©2017 UMBC Training Centers

- character following the matching substring in the input

The LZ family of compression methods is widely used, including popular applications such as Unix zip and PNG images.

LZ77 Coding

The LZ77 coder works by moving a sliding window over the input. This sliding window is an implicit dictionary.

We will describe this sliding dictionary as two parts for ease of explanation. One part will be referred to as the sliding window and the other part as a readahead buffer.

The sliding window has a fixed size of N characters. The window begins at the start of the input and moves from left to right. The end of the window is the cursor position.

The readahead buffer is also of fixed size and begins at the cursor.

As the window slides over the input, a 3-tuple (d, l, c) is output for the longest substring within the readahead buffer that has a match in the sliding window.

> A substring match from the sliding window can extend into the readahead buffer!

Recall that the tuple consists of the following.

d: distance from the current position (cursor) back to the start of a matching substring
l: length of matching substring including any characters forward of the current cursor within some readahead buffer

c: character following the matching substring in the input

If no matching substring from the readhead buffer is found, then the first character in the buffer is encoded with $(0, 0, c)$ where c is again the character in the input after it.

The sliding window is advanced by $l + 1$ so it is just past the last substring with a match in the sliding window.

On completion the input has effectively been partitioned by matched substrings, and each of these has a 3-tuple code. Thus the input is compressed.

©2017 UMBC Training Centers

LZ77 Example

Let's demonstrate LZ77 coding on the simple string, "a a b a a b a a a b a".

Suppose the sliding window and readahead buffer are both four characters in size.

Table 24.1 demonstrates the encoding using zero-indexing.

cursor position (p)	substring match	next char	code (d, l, c)	new cursor position ($p + l + 1$)
0		a	(, a, a)	1
1	a	b	(1, 1, b)	3
3	aaba	a	(3, 4 ,a)	8
8	aba		(4, 3,)	

Figure 24.1: LZ77 coding.

Observe at cursor position $p = 3$ that the sliding window has only three characters.

But a substring match of four characters is made since that is the longest substring in the readahead buffer.

As stated before, the sliding window and readhead buffer make up the sliding dictionary. So any substring can be matched from within this combined length, and therefore a substring match can extend from the sliding window into the readahead buffer.

A match must begin in the sliding window portion but cannot extend

©2017 UMBC Training Centers

past it. As the encoding progresses, the earlier portions that had been coded are no longer visible in the dictionary.

LZ77 Algorithm

The algorithm listing for the LZ77 coding is as follows.

```
Require: SW                          ▷ sliding window buffer of fixed size
Require: RB                          ▷ readahead buffer of fixed size
set $p$ := 0
while $p$ is less than the length of the input do
    From $p$, find the longest substring in RB with a match starting in SW.
    if substring is not null then
        set $d$ := distance to start of matching substring in SW
        set $l$ := length of matching subtring
        set $c$ := next character after matching substring
    else
        set $d, l$ := 0
        set $c$ := next character
    output (d, l, c)
    set $p := p + l + 1$
    advance SW and RB
```

©2017 UMBC Training Centers

Exercise Set 9: Compression

Familiarize yourself with the LZ77 Compression Algorithm, given in the textbook. Attempt to manually compress written sentences using this algorithm before attempting this exercise.

a) Given the name of a `.txt` file (assume the file exists, and that reading it will produce no errors), implement the LZ77 compression algorithm provided in the text. Use a sliding window buffer of size 16, and a readahead buffer of size 8.

©2017 UMBC Training Centers