CSCI2100 Data Structures

Ryan Chan

March 11, 2025

Abstract

This is a note for **CSCI2100 Data Structures** for self-revision purpose ONLY. Some contents are taken from lecture notes and reference book.
Mistakes might be found. So please feel free to point out any mistakes.
Contents are adapted from the lecture notes of CSCI2100, prepared by Irwin King, as well as some online resources.

Contents

Chapter 1

Introduction

1.1 Overview

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

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

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

Algorithm 1.1: Euclid's Algorithm

Data: $m, n \in \mathbb{Z}^+$
Result: $\text{GCD}(m, n)$

while $m > 0$ **do**
  if $n > m$ **then**
    swap m and n
  subtract n from m
return n

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

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

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

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

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

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

1.2 Algorithm

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

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

- Output: At least one quantity is produced.

- Definiteness: Each instruction must be clear and unambiguous.

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

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

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

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

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

1.3 Study of Data

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

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

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

Chapter 2

Analysis

2.1 Complexity

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

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

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

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

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

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

Or we can use the following notation

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

There are some rules to follow:

- Transitivity

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

- Rule of sums

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

- Rule of products

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

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

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

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

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

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

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

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

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

2.2 Recurrence Relations

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

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

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

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

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

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

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

Chapter 3

ADT, List, Stack and Queue

3.1 Abstract Data Type (ADT)

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

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

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

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

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

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

3.2 List

The first abstract data type in this chapter is List.

3.2.1 Definition

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

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

3.2.2 Operations

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

- Create the list and make it empty.

- Determine whether the list is empty or not.

- Determine whether the list is full or not.

- Find the size of the list.

- Retrieve any entry from the list, provided that the list is not empty.

- Store a new entry, replacing the entry at any position in the list, provided that the list is not empty.

- Insert a new entry into the list at any position, provided that the list is not full.

- Delete any entry from the list, provided that the list is not empty.

- Clear the list to make it empty.

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

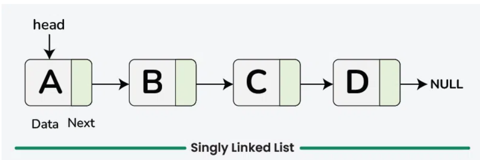
3.2.3 Implementation

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

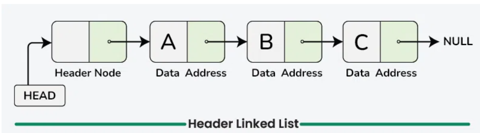
3.2.4 Linked List

There are several types of linked lists:

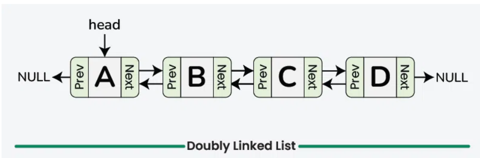
- Singly Linked List



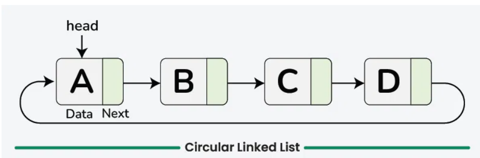
- Singly Linked List with Header



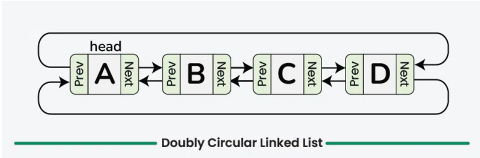
- Doubly Linked List



- Circularly Linked List



- Circularly Doubly Linked List



A polynomial can be represented as

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

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

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

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

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

3.3 Stack

3.3.1 Definition

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

3.3.2 Operations

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

- Create the stack.

- Determine if the stack is empty or not.

- Determine if the stack is full or not.

- Determine the number of entries in the stack.

- Insert (Push) a new entry at one end of the stack, called its top, if the stack is not full.

- Retrieve the entry at the top of the stack, if the stack is not empty.

- Delete (Pop) the entry at the top of the stack, if the stack is not empty.

- Clear the stack to make it empty.

3.3.3 Implementation

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

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

3.3.4 Application

Balance Symbols

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

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

Reverse Polish Calculator

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

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

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

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

3.4 Queue

3.4.1 Definition

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

3.4.2 Operations

Several operations can be performed on a queue:

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

3.4.3 Implementation

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

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

3.4.4 Application

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

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

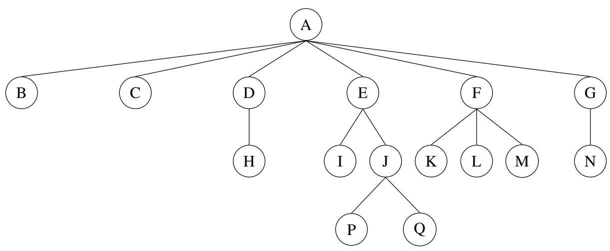
Chapter 4

Trees

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

4.1 General Tree

4.1.1 Nodes



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

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

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

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

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

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

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

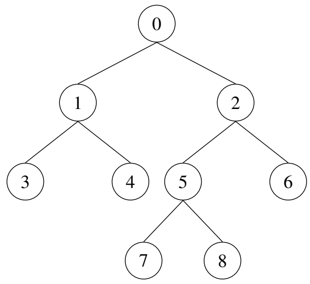
4.1.2 Traversal

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

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

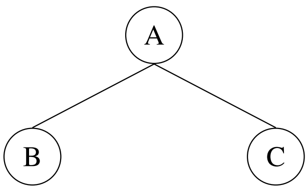
There are three main traversal orders: preorder, inorder, and postorder traversal. However, as long as the traversal follows a systematic way of processing data, it is valid.

For example, in the graph on the right, we have:
- Preorder: 013425786
- Inorder: 314075826
- Postorder: 341785620
- Level-order: 012345678

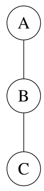


4.2 Binary Tree

4.2.1 Definition



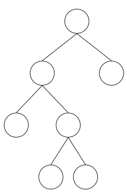
Generic Binary Tree



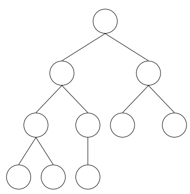
Degenerated Binary Tree

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

4.2.2 Some Binary Trees



Full Binary Tree



Complete Binary Tree

Full Binary Tree

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

Complete Binary Tree

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

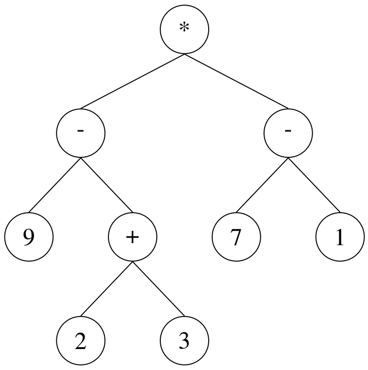
4.2.3 Implementation

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

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

4.3 Expression Tree

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

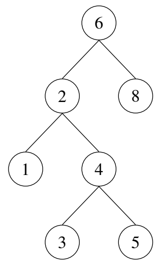


In an expression tree, the leaves represent operands, and the internal nodes represent operators. By using inorder traversal, we can recover the original expression. From the binary tree, using postorder traversal, we can obtain the postfix notation. With the use of a stack, we can implement a calculator,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

4.4 Binary Search Tree

4.4.1 Definition

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



4.4.2 Operations

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

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

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

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

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

4.4.3 Analysis

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

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

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

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

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

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

Which yields

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

■

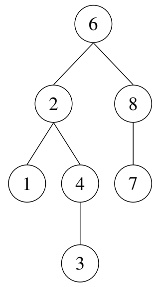
This recurrence gives an average value of $D(n) = O(n \log n)$. Thus, the expected depth of any node should be $O(\log n)$.

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

4.5 AVL Tree

4.5.1 Definition

An AVL (Adelson-Velskii and Landis) tree is a binary search tree with a balancing condition. It is identical to a binary search tree, having the same physical and ordering properties. However, for every node in the AVL tree, the heights of the left and right subtrees can differ by at most 1.
With an AVL tree, all tree operations, except insertion, can be performed in $O(\log n)$ time.
To construct the smallest AVL tree of height n, we can use the two smallest AVL subtrees of heights $n-1$ and $n-2$. By doing so recursively, we can find the smallest AVL tree.



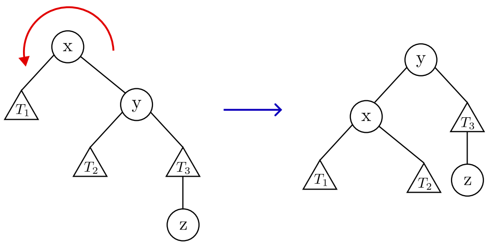
The height of an empty tree is defined to be -1. Height information is kept for each node. The height of an AVL tree is at most roughly $1.44\log(n+2) - 0.328$, but in practice, it is about $\log(n+1) + 0.25$.

4.5.2 Operations

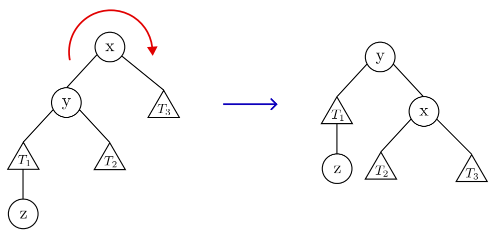
All tree operations can be performed in $O(\log n)$ time, except possibly insertion. Insertion and deletion operations need to update the balancing information since they might violate the AVL tree property. Therefore, we need to restore the property by means of rotations.

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

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



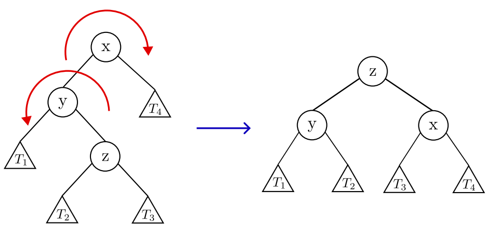
Left Rotation



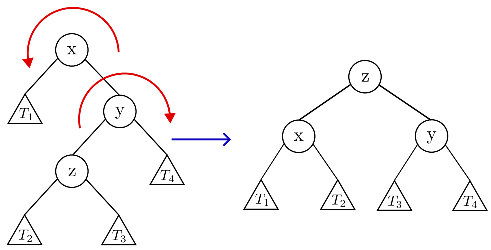
Right Rotation

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

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



Left-Right Rotation

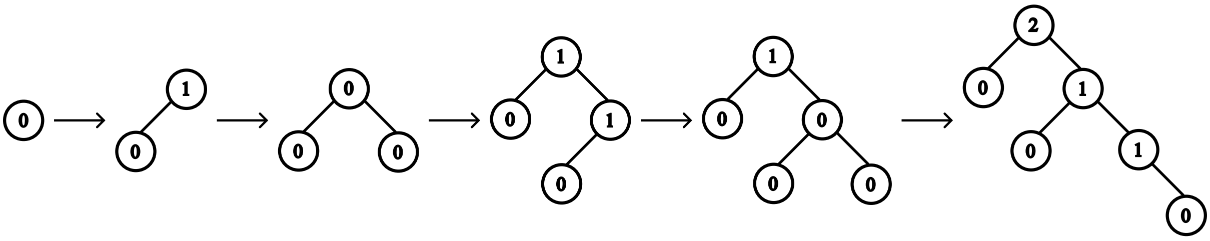


Right-Left Rotation

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

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

For example, the method for checking imbalance is demonstrated below:



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

4.6 B-Tree

4.6.1 Definition

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

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

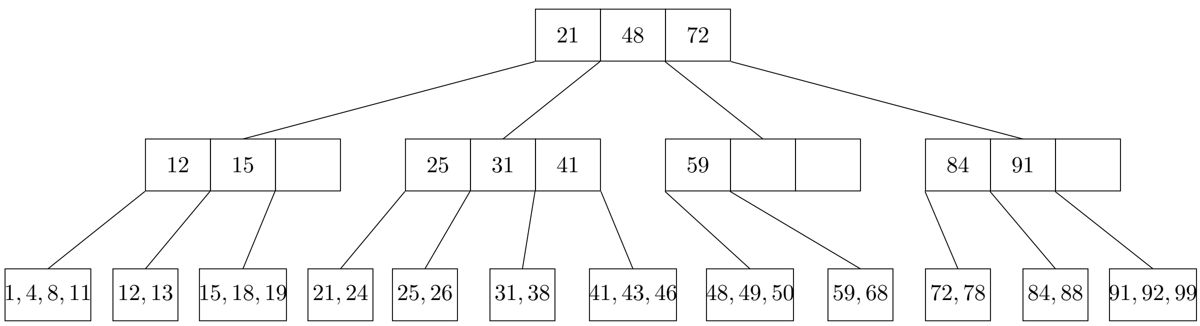
- The non-leaf nodes store up to $m - 1$ keys.

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

- All leaves are at the same depth and have between $\lceil \frac{l}{2} \rceil$ and l elements.

- All data is stored at the leaves. Each interior node contains pointers $P_1, P_2, \ldots, P_m$ to its children, and values $K_1, K_2, \ldots, K_{m-1}$, where each K_i represents the smallest key found in the subtrees $P_2, P_3, \ldots, P_m$, respectively.

For every node, all the keys in subtree P_1 are smaller than the keys in subtree P_2, and so on.



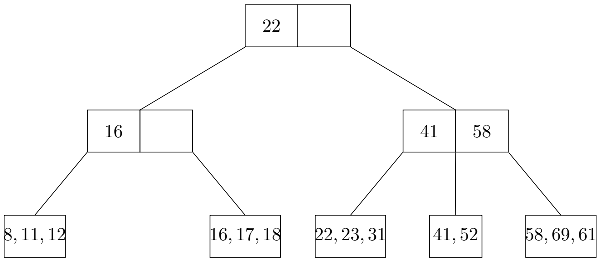
B-tree of order 4

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

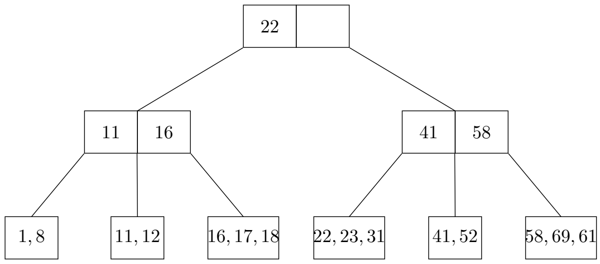
4.6.2 Operations

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

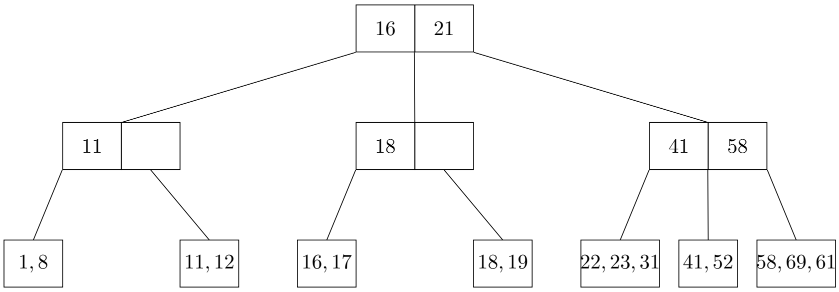
To insert 1, we find the location. However, since the leftmost node is full, we cannot insert it there. This can be solved by splitting the node into two nodes with two keys, then adjusting the information of the parent. Next, we try to insert 19. Since the rightmost node is full, we need to split it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



Original Tree



Insert 1



Insert 19

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

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