Exploring the Data Structures and Algorithms behind Version Control Systems

(Student Proposed)

Reece Donovan

Final Year Project
B.Sc. Computer Science

Supervisor: Prof. Ken Brown
Second Reader: Dr. Klaas-Jan Stol

Department of Computer Science
University College Cork

April 2023

Abstract

Declaration of Originality

Acknowledgements

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 What is a Version Control System

A Version Control System (VCS) saves modifications made by individual software developers, allowing them to track their work over time and making the process more accessible. Furthermore, it facilitates sharing data between nodes, where each node can be kept up to date with the latest versions, minimising the need to handle merge conflicts.

Version Control Systems provide several advantages, such as aiding in collaboration among programmers and improving efficiency when working on large groups of mixed products. In addition, VCSs offer an audit trail and easy branching functionality, simplifying team-based development. It also enables an understanding of who has worked on specific pieces or sections during a given period.

By ensuring all changes are appropriately tracked, VCSs promote transparency within teams while maintaining optimal performance levels through streamlined management processes. This helps avoid errors arising from inconsistencies between versions, such as mismatched documents or programming problems resulting from conflicting edit operations.

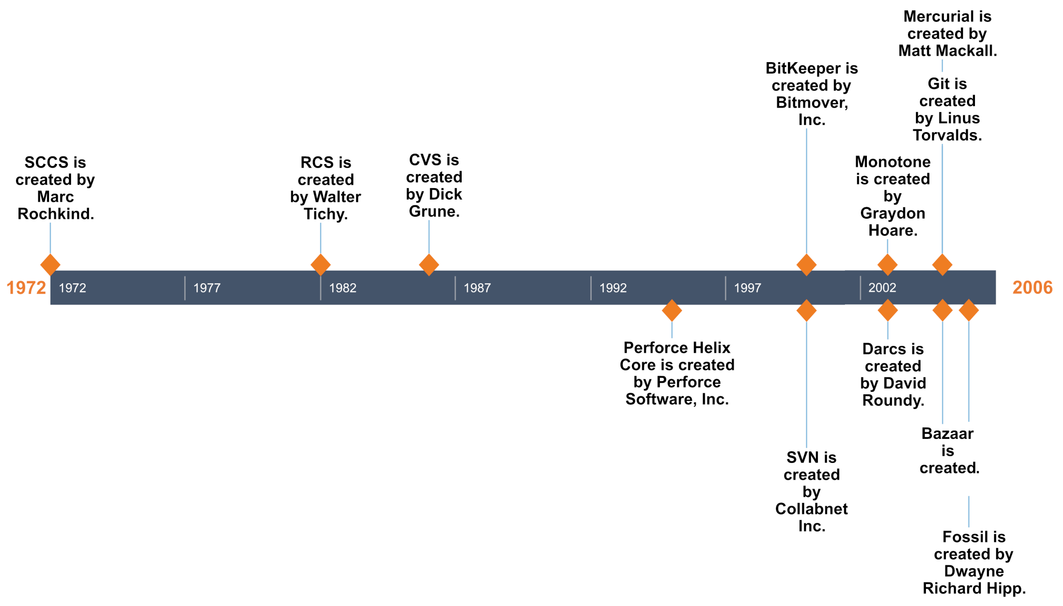
Significance of Version Control Systems

For most software projects, the source code is a valuable asset that must be protected. Additionally, for many software teams, the source code represents a repository of invaluable knowledge and understanding about the problem that developers have amassed and refined through meticulous effort.

Version control safeguards this precious resource by preventing catastrophes, casual degradation, human error, or unintended consequences from affecting its quality. Software developers working in teams continually write new source code and modify existing code, which is organised into file folders called "file trees". Developers may work on different tasks simultaneously, such as implementing new features or fixing unrelated bugs, by altering parts of the file tree.

Version control assists teams in addressing these challenges by tracking each change made by contributors, helping to prevent concurrent work from conflicting with one another. Consequently, any incompatibility can be detected and resolved without impeding team members' progress further down the line, ensuring that changes do not introduce bugs.

2.2 Evolution of Version Control Systems



1972
2006
1972
1977
1982
1987
1992
1997
2002
SCCS is created by Marc Rochkind.
RCS is created by Walter Tichy.
CVS is created by Dick Grune.
BitKeeper is created by Bitmover, Inc.
Mercurial is created by Matt Mackall.
Git is created by Linus Torvalds.
Monotone is created by Graydon Hoare.
Perforce Helix Core is created by Perforce Software, Inc.
SVN is created by Collabnet Inc.
Darcs is created by David Roundy.
Bazaar is created.
Fossil is created by Dwayne Richard Hipp.

Figure 2.1: Timeline of the Creation of Version Control Systems [5]

2.2.1 Local Version Control Systems (LVCS)

Local Version Control Systems are systems designed to operate on a single machine, early VCSs were intended to track changes for individual files, and checked-out files could only be edited locally by one user at a time [5]. In addition, they were built on the assumption that all users would log into the same shared Unix host with their own accounts, which was not always possible.

The main benefit of LVCS is that it provides a basic level of version control functionality without requiring any network connectivity. However, this can also be seen as a drawback when you consider that with LVCS, there is no central repository to store the code, which means that each developer has their own copy. This can lead to issues with merging changes and tracking changes across multiple copies.

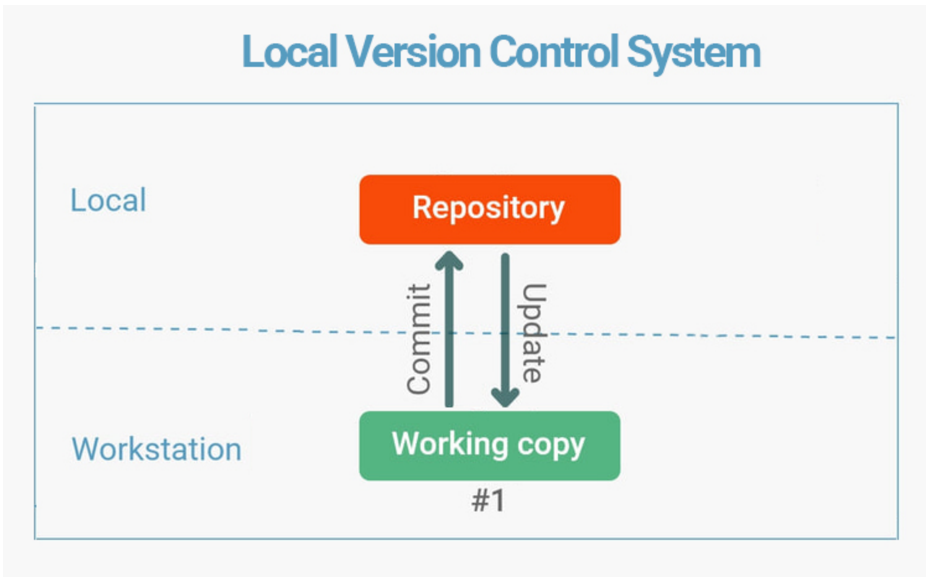


Figure 2.2: Local Version Control System (LVCS)

Source Code Control System (SCCS)

Source Code Control System was released in 1972 and is one of the first successful VCS tools [5]. It was written by Marc Rochkind at Bell Labs, who wanted to solve the problem of tracking file revisions. The tool made tracking down bugs introduced into a program significantly more manageable. SCCS is worth understanding at a basic level because it helped set up modern VCS tools that developers use today.

Architecture

Much like modern VCS tools, SCCS has a set of commands that allow developers to work with the versioning of files. The basic command functionality is:

- Check-in files to track their history.
- Check-out specific file versions for review.

- Check-out specific file versions for editing.
- Check-in new file versions with comments explaining the changes.
- Revert changes made to a checked-out file.
- Basic branching and merging of changes.
- Print a log of a file's version history.

A particular type of file called an `s-file` or a `history file` is created when a file is tracked by SCCS. This file is named with the original filename prefixed with an `s.` and is stored in a subdirectory called `SCCS`.

So a file called `test.txt` would get a history file created in the `./SCCS/` directory with the name of `s.test.txt`. When created, the `s-file` contains the original file contents, a header that contains the file's version number, and some other metadata. There are also checksums stored in the `s-file` that are used to verify the integrity of the file (i.e. to ensure that the file has not been tampered with). The `s-file` content is not encoded or compressed in any way, which is a clear difference from modern VCS tools.

Since the original file's content is now stored in the history file, it can be retrieved into the working directory for review, compilation, or editing. Further changes made to this new copy, such as line additions, modifications and removals, can be checked back into a revised version of the history file, which increments its revision number [5].

Subsequent SCCS check-ins only store only deltas or changes to a file instead of storing entire contents each time; when a check-in is made, subsequent revisions are added onto existing delta tables inside an amended history file (history files do not use compression). This decreases the size of these large histories since they are not using compression on their files, so they take up more space than just having one complete copy that you are tracking with no differences like Word docs etc.

As previously mentioned, SCCS uses the Delta method known as Interleaved Deltas, which allows constant-time checkout regardless of how old your checked-out revision is - i.e., older revisions do not take longer than newer ones.

It is important to note that all files are tracked and checked in separately; there is no way to check in changes on multiple files as a part of one atomic unit - like a commit in Git. Each tracked file has its corresponding history file, which stores revisions. Generally, this means that the version numbers of different files cannot match each other. However, matching revision numbers can be achieved by editing every file at once (even if not all of the changed areas have fundamental changes) and checking them all together. This will increment revision numbers for any modified content, so they will be consistent with their revisions—but it's not comparable to including lots of different pieces within a single commit like you would in Git. In SCCS, these make individual check-ins across separate history folders rather than one extensive report containing everything at once.

When a file is checked out for editing in SCCS, it is locked so that anyone else cannot edit it. This prevents changes from being overwritten by other users and limits development since only one user can edit the file simultaneously.

The software supports branches that store sequences of changes to specific files - these can then be merged back into the original versions or with copies of other branched versions of the same parent branch.

Revision Control System (RCS)

The Revision Control System was released in 1982 as an alternative to the closed-source Source Code Control System. Developed by Walter Tichy and written in C, RCS was released under the GNU General Public License, making it suitable for use in open-source projects.

> "RCS manages revisions of text documents, in particular source programs, documentation, and test data. It automates the storing, retrieval, logging and identification of revisions." – Walter Tichy [7]

Architecture

RCS shares many traits with its predecessor[5], including:

- Handling revisions on a file-by-file basis.
- Changes across multiple files can't be grouped together into an atomic commit.
- Tracked files are intended to be modified by one user at a time.
- No network functionality.
- Revisions for each tracked file are stored in a corresponding history file.
- Basic branching and merging of revisions within individual files.

Upon checking a file into RCS for the first time, a corresponding history file is created in the local `./RCS/` directory. The history file is postfixed with a `,v`, so a file named `test.txt` would be tracked by a file called `test.txt,v`[5].

RCS employs a reverse-delta scheme for storing file changes. When a file is checked in, the history file contains a complete snapshot of its contents. Subsequent modifications and check-ins result in RCS calculating a single delta—the difference between the new version of that specific revision and the previously recorded version - and saving it along with an older snapshot if necessary.

The reverse-delta scheme functions by checking out an earlier revision from the newest version and applying consecutive deltas until reaching the desired revision. Starting from the newest version allows quick checkout times, as the current revisions' snapshots are readily accessible.

However, when attempting to access older versions beyond just one recent update (e.g., 1 or 2 old versions), the process becomes considerably more complex. This is because these older versions' snapshots must be calculated against each other before they can be applied to the newest version.

Unlike RCS, SCCS maintains a consistent checkout time for any revision. Additionally, RCS history files do not store checksums, meaning file integrity cannot be guaranteed.

2.2.2 Centralized Version Control Systems (CVCS)

A Centralised Version Control System is a system that enables developers to work together on the same project by storing the primary copy of files in a central repository. This system keeps track of all files and saves information in the local repository. CVCS is called centralised because there is only one central server or repository.

The server maintains a complete record of issues, while clients only maintain a local copy of the shared documents. All developers make their modifications to the repository through checkout. However, only the last version of the files is retrieved from the server, meaning any modifications made will automatically be shared with other developers.

Users can modify in parallel with their local copy of shared documents and sync with the central server to release their contributions and make them visible to other collaborators. However, because centralised version control systems rely on one repository that includes the correct project version, they must restrict write access so that only trusted contributors can commit modifications.

CVCS has some challenges, such as if the central server is inaccessible, users cannot merge their work or save the released modifications. Also, if the central repository is corrupted, everything will be lost. Contributors must be the ones who have writing permissions to perform basic tasks, such as reverting modifications to a previous state, creating or merging branches, or releasing modifications with complete revision history. This limitation affects participation and authorship for new contributors.

So, the main drawbacks of using CVCS are that it requires a network connection to work on the source code, developers must order to contribute to a project, and a single point of failure is an issue when using one server.

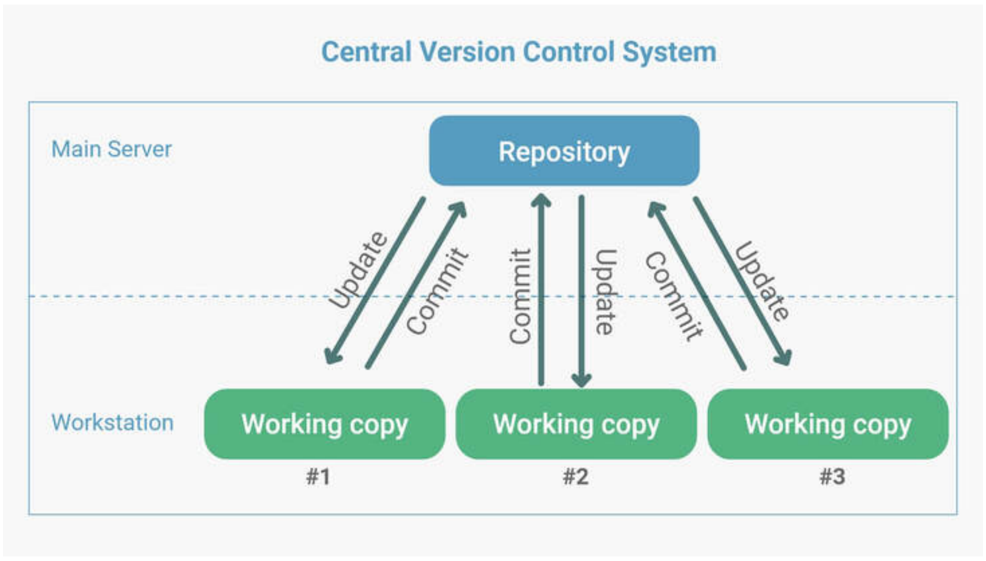


Figure 2.3: Centralized Version Control System (CVCS)

Concurrent Versions System (CVS)

Dick Grune developed the Concurrent Versions System in 1986 to introduce a networking element to version control. Written in `C`, CVS became the first widely used VCS tool that enabled multiple users to work on the same project simultaneously from different locations. This innovation began the second generation of VCS tools and facilitated collaboration among geographically dispersed development teams.

Architecture

CVS functions as a frontend for RCS, providing a set of commands to interact with files in a project while utilising the RCS history file format and commands behind the scenes. CVS allowed multiple developers to check out and work on duplicate files for the first time in history by employing a centralised repository model.

When a project is imported into CVS, each file is converted into a `,v` history file and stored in a central directory referred to as a `module`[5]. Generally, the repository resides on a remote server accessible via a local network or the Internet.

A developer checks out a copy of the module, which is copied to a working directory on their local machine. No files are locked during this process, allowing an unlimited number of developers to check out the module simultaneously. In addition, developers can modify their checked-out files and commit changes as needed.

When a developer commits a change, other developers must update their working copies through a (usually) automated merge process before committing their changes. Occasionally, merge conflicts require manual resolution before a commit can be made. Therefore, CVS also offers the capability to create and merge branches.

Perforce Helix Core

Perforce Helix Core is a proprietary VCS developed, owned, and maintained by Perforce Software Inc., Written in `C` and `C++`; it was initially released in 1995. Although primarily designed for a centralised model, it also offers a distributed model option. Helix Core is commonly used by large companies managing substantial content with large binary files, such as in the video game development industry. Although typically cost-prohibitive for smaller projects, Perforce provides a free version for teams of up to five developers.

Architecture

Perforce Helix Core utilises a server/client model. The server, a process called `p4d`, listens for incoming client connections on a designated port, typically port `1666`. The client, a process called `p4`, is available in both command-line and GUI variants. Users run the `p4` client to connect to the server and issue commands. Support for various programming language APIs, including Python and Java, enables automated issuance and processing of Helix Core commands via scripts. Integrations are also available for IDEs like Eclipse and Visual Studio, allowing users to work with version control within those tools.

The Helix Core Server manages repositories called depots, which store files in directory trees similar to Apache Subversion (SVN). Clients can check out sets of files and directories from the depots into local copies called `workspaces`. The atomic unit used to group and track changes in Helix Core depots is called the `changelist`, which is analogous to Git commits. In addition, Helix Core implements two similar forms of branching: `branches` and `streams`. While branches represent separate lines of development history, a stream is a branch with added functionality that Helix Core uses to provide recommendations on best merging practices throughout the development process.

When a file is added for tracking, Helix Core classifies it using a `file type` label. The two most commonly used file types are text and binary. For binary files, the entire file content is stored each time the file is saved. This is a common VCS tactic for handling binary files, which are not amenable to the normal merge process, as manual conflict resolution is typically impossible.

Only the deltas (changes between revisions) are stored for text files. Text file history and deltas are stored using the Revision Control System (RCS) format, which tracks each file in a corresponding `,v` file in the server depot. This is similar to Concurrent Versions System (CVS), which also leverages RCS file formats for preserving revision history. Files are often compressed using gzip when added to the depot and decompressed when synced back to the workspace.

Apache Subversion (SVN)

Subversion was created in 2000 by CollabNet Inc. and is now maintained by the Apache Software Foundation. Written in `C`, it was designed to offer a more robust centralised solution than Concurrent Versions System (CVS)[5].

Architecture

Similar to Concurrent Versions System (CVS), Subversion employs a centralised repository model, requiring remote users to have a working network connection to commit their changes to the central repository. However, in contrast to CVS, where a commit operation could fail midway due to a network outage and leave the repository in a corrupted and inconsistent state, Subversion ensures the repository remains consistent. Additionally, a Subversion commit or revision can include multiple files and directories, enabling users to track sets of related changes together as a grouped unit, as opposed to previous storage models that tracked changes separately for each file.

Subversion utilises a storage model called `FSFS (File System atop the File System)` for tracked files. The model's name originates from its database structure, which mirrors the operating system's file and directory structure. However, subversion's unique feature is its filesystem design, which tracks not only files and directories but also different versions of these files and directories as they change over time. This creates a filesystem with an added time dimension. Moreover, Subversion treats folders as first-class citizens, allowing empty folders to be committed, unlike other systems such as Git, where empty folders go unnoticed.

When a Subversion repository is created, a (nearly) empty database of files and folders is established [5]. Next, a directory called `db/revs` is created to store all revision tracking information for the committed files. Each commit, which may include changes to multiple files, is stored in a new file in the `revs` directory, named with a sequential numeric identifier starting with `1`. When a file is committed for the first time, its entire content is stored. Subsequent commits of the same file store only the changes, also known as `diffs` or `deltas`, to save space. Additionally, `deltas` are compressed using `lz4` or `zlib` compression algorithms to reduce their size further.

By default, this approach is only employed to a certain extent. Although storing file deltas instead of the entire file each time saves storage space, it adds time to checkout and commit operations, as all the deltas must be combined to recreate the file's current state. For this reason, Subversion stores up to 1023 deltas per file before saving a new complete copy of the file, striking a balance between storage and speed.

2.2.3 Distributed Version Control Systems (DVCS)

Distributed Version Control Systems (DVCS) were developed to address the limitations of Centralized Version Control Systems (CVCS), enabling more effective branching and merging, seamless local VCS operations, and improved collaboration among developers. Due to these limitations associated with Centralized Version Control Systems (CVCS), Distributed Version Control Systems have become widely adopted in Open Source Software (OSS) projects.

DVCS is designed to function in two ways: it maintains file histories locally on each device and can synchronise local user modifications with servers when necessary, enabling the sharing of these modifications with others. In DVCS, developers can work independently or collaboratively on the same project, as they have access to all necessary repositories. In addition, any repository can be cloned from another, meaning no repository holds greater importance than others.

In response to the need for more advanced versioning of software artefacts, several Distributed Version Control Systems, such as Mercurial, and Git, emerged in the software field. Many OSS projects have widely adopted these tools. DVCS operations are significantly faster than those in CVCS, as they are performed locally, while CVCS operations require remote connections. Some experts believe that distributed systems will eventually replace centralised ones due to their suitability for more extensive projects involving independent developers seeking full functionality, even without a network connection. In addition, distributed systems offer advantages such as saving earlier drafts of work without publicly releasing or sharing them with others.

Collaboration between team members and the ability for individual developers to serve as servers or clients are essential features of version control systems, allowing developers to work on source code without being connected to a central or remote repository.

However, DVCS introduces some challenges, including the lack of a coherent version numbering system due to the absence of a centralised versioning server. Instead, DVCS relies on hash modifications or a unique GUID. This lack of a central server also complicates system backup.

The two most common criticisms of DVCS are the unavailability of pessimistic locks and weak support for binary files. Despite these disadvantages, the transition from centralised to decentralised version control systems continues, as developers can work offline and efficiently work incrementally. This flexibility enables developers to assume multiple roles, such as developing new tasks or fixing errors, leading to exploratory coding and greater freedom in their workflow while maintaining control over their code's release schedule.

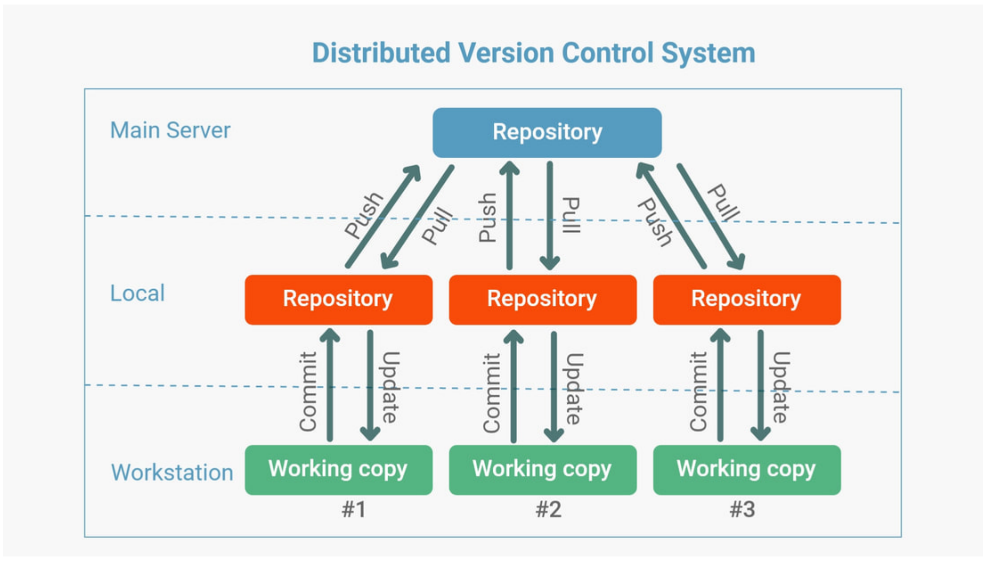


Figure 2.4: Distributed Version Control System (DVCS)

Git

Git, created in 2005 by Linus Torvalds (also the creator of Linux), is primarily written in C with some shell scripts. Git was initially developed for the Linux codebase and has since become the most popular VCS in use today due to its features, flexibility, and speed. Torvalds explains that Git is a robust set of tools with many options, and its usage is often determined by what works best for collaboration rather than technical limitations.

> ”You can do a lot of things with Git, and many of the rules of what you should do are not so much technical limitations but are about what works well when working together with other people. So Git is a very powerful set of tools, and that can not only be overwhelming at first, it also means that you can often do the same (or similar) things different ways, and they all ’work.’” – Linus Torvalds [1]

Git repositories are commonly hosted on local servers and cloud services, forming the backbone of a broad set of DevOps tools available from popular service providers, including GitHub, BitBucket, GitLab, and many others [5].

Architecture

As a Distributed Version Control Systems (DVCS), Git ensures that no repository copy needs to be designated as the centralised copy—instead, all copies are created equal. This contrasts with second-generation VCS, which relies on a centralised copy for users to check in and out.

This design allows developers and coding partners to share changes directly with each other before merging their changes into an official branch, fostering a flexible distributed workflow for team collaboration.

Moreover, developers can commit changes to their local copy of the repository without other repositories knowing about it. This enables commits without a network or internet connection, allowing developers to work offline until they are ready to push their changes to other repositories for review, testing, or deployment.

When a file is added for tracking with Git, it is compressed using the `zlib` compression algorithm and hashed using a `SHA-1` hash function [5]. This generates a unique hash value corresponding to the file content, which Git stores in an object database located in the hidden `.git/objects` folder. These files, called `Git blobs`, are created each time a new file (or a changed version of an existing file) is added to the repository.

Git uses a staging index that functions as a temporary space for changes being readied for a commit. When changes are set to be committed, their compressed data is referenced in a unique index file, appearing as a tree object. `Trees` in Git link blob objects to actual file names, file permissions, and connections to other trees, signifying the status of a specific collection of files and directories. After all associated changes have been staged for commit, the index tree can be committed to the repository, generating a commit object within the Git object database [5].

A commit denotes the primary tree for a specific revision, and the commit author, email address, date, and a descriptive commit message. Each commit also retains a reference to its preceding commit/commits, constructing a record of the project's evolution.

Git objects, including blobs, trees, and commits, are all compressed, hashed, and saved in the object database based on their respective hash values. These standalone objects avoid using diffs for space conservation, making Git highly efficient since the complete content of every file revision is readily available as an individual object.

Nonetheless, particular operations, such as pushing commits to a remote repository, storing an excessive number of objects, or manually executing Git's garbage collection command, may prompt Git to reorganise objects into pack files. This packing procedure compresses inverse diffs to remove duplicate content and minimise size. This leads to the creation of `.pack` files containing the object data, each paired with a corresponding `.idx` (index) file that references the packed objects and their positions within the pack file. These pack files are transmitted across the network when branches are pushed to or fetched from remote repositories. When pulling or retrieving branches, the pack files are decompressed to generate loose objects in the object repository.

Mercurial

Mercurial, created in 2005 by Matt Mackall and written in `Python`, initially aimed to host the codebase for Linux, but Git was chosen instead [5]. As the second most popular distributed VCS after Git, Mercurial is used far less frequently.

Architecture

Comparable to Git, Mercurial is a Distributed Version Control Systems (DVCS) that enables multiple developers to work on seperate copies of the same project. Although Mercurial utilises many similar technologies as Git, such as compression and `SHA-1` hashing, it does so in distinct ways.

When a new file is tracked in Mercurial, a corresponding `revlog` (revision log) file is generated for it in the hidden `.hg/store/data/` directory. `Revlog` files can be viewed as advanced versions of the history files employed by older VCSs such as Concurrent Versions System (CVS), Revision Control System (RCS), and Source Code Control System (SCCS).

Unlike Git, which generates a new blob for each version of every staged file, Mercurial merely creates a new entry in the revlog for that file. To conserve space, each new entry contains only the delta (modifications) from the preceding version. Once a certain number of deltas is achieved, a complete snapshot of the file is stored again, minimising lookup time when applying numerous deltas to reconstruct a specific file revision.

These file revlogs are named to correspond with the files they monitor but are postfixed with `.i` and `.d` extensions. The `.d` files hold the compressed `delta` content, while the `.i` files function as `indexes` to locate distinct revisions within the `.d` files swiftly. For small files with few revisions, both the `indexes` and `content` are stored in `.i` files. Revlog file entries are compressed for efficiency and hashed for identification, with the hash values referred to as `nodeids`.

2.3 Summary of Key Features

2.3.1 Repositories

2.3.2 Commits

2.3.3 Branching and Merging

2.3.4 Pulling and Pushing

Chapter 3

Design

In this chapter, we examine various data structures and algorithms with key aspects that may make them suitable for incorporation into a `Version Control System`.
We also assess what metrics are relevant to facilitate the comparison of these data structures and algorithms.

3.1 Data Structures

Data structures are objects that can be used to store, organize, and manipulate large amounts of data. They play a crucial role in computer science and are fundamental building blocks of many software systems, including `Version Control Systems`. The choice of data structure is critical to the overall performance and scalability of a `Version Control System`.

In order to evaluate the suitability of a data structure for implementation in a `Version Control System`, it is necessary to consider several core aspects. Firstly, `operational efficiency`, which refers to the time and space complexity of the basic operations performed by the data structure, is a crucial consideration. The efficiency of these operations can have a significant impact on the overall performance of the `Version Control System`.

Another important aspect is the data structure's `structural specificity`, which encompasses how data is stored and organized within the structure. This is critical because the structural specifics can affect the ease of implementation and the ability to efficiently perform operations such as `insertions`, `deletions`, and `updates`.

Lastly, the `implementation details` must be considered, including the ease of implementation and compatibility with the programming language. A data structure that is straightforward to implement and easy to maintain/iterate upon will result in a more streamlined and efficient `Version Control System`.

3.1.1 Linked List

A `Linked List` is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. The `Linked List` is the most sought-after data structure when it comes to handling dynamic data elements [3].

There are two main types of `Linked Lists`: `Singly-Linked Lists (SLL)` and `Doubly-Linked Lists (DLL)`, but we will only be considering the `Doubly-Linked List (DLL)` when we reach the `Implementation` chapter.

Singly-linked lists (SLL)

- `SLL` nodes contain two fields: `data` field and `next` pointer field.
- Traversal of a `SLL` can be done using the `next` pointer field only. Meaning, the `SLL` can be traversed in only one direction, from the first node to the last node.
- The `SLL` occupies less memory than a `DLL` because it does not contain a `prev` pointer field.
- `SLL` is preferred over `DLL` when it comes to the execution of stack and queue operations.
- `SLL` is also preferred over `DLL` to save memory when a searching operation is not required.

Table 3.1: Efficiency Analysis of Singly-Linked List Operations

Operation	Worst Case	Average Case
Access	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$
Insert (at Head)	$O(1)$	$O(1)$
Insert (at Current)	$O(1)$	$O(1)$
Insert (at Tail)	$O(1)$	$O(1)$
Delete	$O(n)$	$O(n)$
Delete (at Head)	$O(1)$	$O(1)$
Delete (at Current)	$O(n)$	$O(n)$
Delete (at Tail)	$O(n)$	$O(n)$

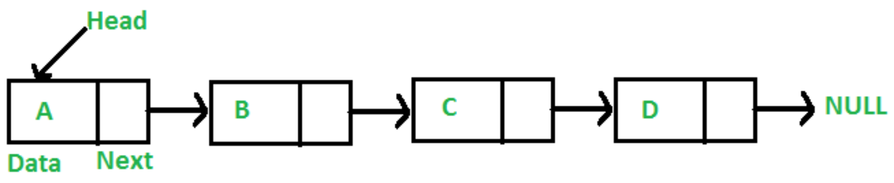


Figure 3.1: Singly Linked List (SLL) [8]

Doubly-linked lists (DLL)

- `DLL` nodes contain three fields: `data` field, `prev` pointer field and `next` pointer field.
- Traversal of a `DLL` can be done using the `next` pointer field or the `prev` pointer field. Meaning, the `DLL` can be traversed in both directions, from the first node to the last node and vice versa.
- The `DLL` occupies more memory than a `SLL` because it contains a `prev` pointer field.

Table 3.2: Efficiency Analysis of Doubly-Linked List Operations

Operation	Worst Case	Average Case
Access	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert	$O(n)$	$O(n)$
Insert (at Head)	$O(1)$	$O(1)$
Insert (at Current)	$O(1)$	$O(1)$
Insert (at Tail)	$O(1)$	$O(1)$
Delete	$O(n)$	$O(n)$
Delete (at Head)	$O(1)$	$O(1)$
Delete (at Current)	$O(1)$	$O(1)$
Delete (at Tail)	$O(1)$	$O(1)$

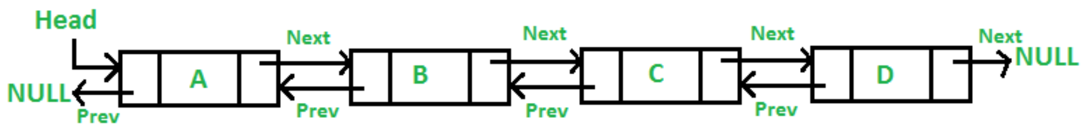


Figure 3.2: Doubly Linked List (DLL) [8]

Advantages

- Dynamic size: As new versions of a file are created, the `Linked List` can grow in size to accommodate the new data, without the need to pre-allocate memory.
- Efficient storage of file changes: Each node in the `Linked List` can store the entire contents of a file version, along with metadata such as the date and time the change was made. This allows for efficient storage of file changes over time.
- Easy traversal of file history: The `Linked List` structure allows for easy traversal of the file history, as each node contains a reference to the previous version of the file. This makes it easy to track changes and revert to previous versions of a file.
- Memory efficient: The `Linked List` structure is memory efficient, as each node only contains the current version of the file and changes made to it, along with simple references to the next and previous nodes in the list. This means that the `Linked List` structure does not need to store the entire file history in memory, which can be a significant amount of data.

Disadvantages

- Inefficient retrieval of specific file versions: Retrieving a specific version of a file can be slow, as the `Linked List` structure does not allow for random access to the file history. This means that the entire file history must be traversed from the most recent version of the file to the desired version.
- Limited scalability: For large file histories, the `Linked List` structure can be less efficient and will not scale as well as other data structures.
- Extra memory overhead: Each node in the `Linked List` structure contains a reference to the next and previous nodes in the list, which can add up when dealing with large file histories.

- Not suitable for concurrent access: The `Linked List` structure is not suitable for concurrent access, as it is not thread-safe and can lead to data corruption and race conditions.

Implementation Details Exercitation ullamco culpa velit excepteur aute esse amet. Quis adipisicing consequat quis sunt elit cupidatat sunt ipsum nostrud laborum aliqua veniam veniam commodo. Minim ullamco aute aliquip eiusmod officia cillum fugiat magna consectetur aute sunt aliqua labore. Reprehenderit dolore commodo deserunt laborum culpa laborum elit. Labore Lorem quis culpa amet adipisicing pariatur consequat eu proident in officia aute voluptate. Excepteur irure deserunt et ullamco deserunt labore anim dolor amet est est culpa. Magna eiusmod nulla ipsum esse anim nostrud mollit fugiat proident magna laboris.

Ut non aliquip aliquip Lorem reprehenderit nisi qui aliqua cupidatat enim adipisicing deserunt. Eiusmod est dolore ut ipsum Lorem et sunt est minim in Lorem. Reprehenderit ea proident officia anim dolore incididunt sunt labore sunt. Ea sunt incididunt anim tempor. Esse amet ut magna id irure ex.

Id do cillum ad ad officia dolor sunt deserunt amet ex. Pariatur aute sint anim aute id irure reprehenderit laboris non tempor id. Labore in ad sunt nulla dolore velit ut aliquip amet dolore quis voluptate nisi. Magna non magna nulla officia magna voluptate officia dolore Lorem ea sunt duis. Non dolore proident voluptate consequat consequat laborum aliqua veniam et occaecat amet ea dolore.

Summary A `Doubly-Linked List` is a data structure that consists of a sequence of nodes, each node having a data field and two pointers, one pointing to the next node in the list and the other pointing to the previous node.

One of the main advantages of using a `Doubly-Linked List` is that it allows for easy insertion and deletion of nodes, making it possible to add new file versions or remove outdated ones easily.

However, Concurrent access to a `Doubly-Linked List` can lead to data inconsistencies and race conditions, and proper synchronization must be implemented to prevent these issues.

3.1.2 Binary Tree

A `Binary Tree` is a hierarchical data structure that consists of a set of nodes, where each node can have at most two children. The topmost node in the tree is called the root node, and the nodes that do not have any children are called leaf nodes. The nodes that have children are called internal nodes. The `Binary Tree` structure is a special case of the `Tree` data structure, where each node can have at most two children.

Each node in a `Binary Tree` contains the following elements:

1. Data: The data stored in the node.
2. Left child: A pointer to the left child node.
3. Right child: A pointer to the right child node.

There are several different types of `Binary Tree` structures, including:

1. Full Binary Tree: A `Binary Tree` where each node has either zero or two children.
2. Complete Binary Tree: A `Binary Tree` where all levels of the tree are completely filled, except for the last level. The last level of the tree is filled from left to right.
3. Balanced Binary Tree: A `Binary Tree` where the difference between the height of the left and right subtrees of any node is not greater than one.
4. Degenerate (or Pathological) Binary Tree: A `Binary Tree` that is not balanced, where the height of the tree is equal to the number of nodes in the tree.

Table 3.3: Efficiency Analysis of Binary Tree Operations

Operation	Worst Case	Average Case
Search	$O(n)$	$O(\log n)$
Insert	$O(n)$	$O(\log n)$
Delete	$O(n)$	$O(\log n)$

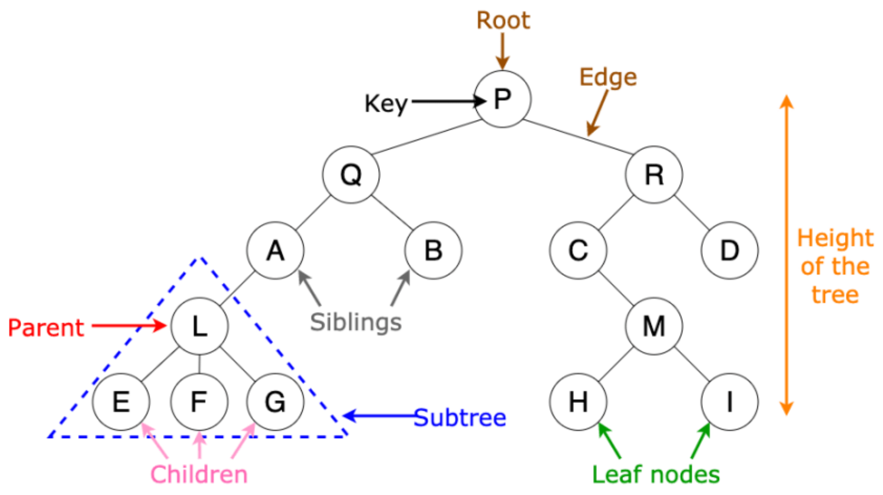


Figure 3.3: Binary Tree [2]

Advantages

- Efficient storage of file changes: Each node in the `Binary Tree` structure can store the entire contents of a file version, along with metadata such as the date and time the version was created. This allows for efficient storage of file changes, as the entire file history can be stored in a single `Binary Tree` structure.
- Easy traversal of file history: The `Binary Tree` structure allows for easy traversal of the file history, with the root node representing the most recent version of the file and the leaf nodes representing the oldest versions of the file. This makes it easy to track changes and revert to previous versions of the file.
- Flexibility: `Binary Tree` structures can be used to implement a variety of other data structures, such as `Binary Search Trees`, `AVL Trees`, `Heaps`, and others, which can be useful for other operations in a `Version Control System`.

Disadvantages

- Inefficient retrieval of specific file versions: The `Binary Tree` structure can be slow when retrieving specific file versions, as it requires traversing the `Binary Tree` to find the desired node. This can be inefficient when dealing with large file histories.
- Limited scalability: For large file histories, the `Binary Tree` structure can be less efficient and may not scale as well as other data structures.

- Extra memory overhead: Each node in the `Binary Tree` structure requires extra memory to store the pointers to the left and right child nodes. This can become significant when dealing with large file histories.
- Unbalanced trees can lead to poor performance: If the `Binary Tree` structure becomes skewed, the time complexity of searching, insertion, and deletion operations can become `O(n)`, where `n` is the number of nodes in the tree.
- Not suitable for concurrent access: The `Binary Tree` structure is not suitable for concurrent access, as it is not thread-safe. This can lead to data corruption and race conditions.

Implementation Details Nulla cillum laborum quis et cillum eu. Id exercitation ad aliquip ipsum elit excepteur tempor occaecat enim excepteur culpa aliqua ullamco pariatur. Et do elit nisi duis et aliquip consequat dolor labore.

Laborum consequat elit fugiat excepteur esse exercitation anim anim est eiusmod aliquip ad. Labore veniam cillum officia aute elit minim minim in laboris. Incididunt velit amet consequat officia nulla exercitation ex voluptate in duis ullamco Lorem.

Nostrud officia consectetur proident aliquip elit commodo do pariatur eu aliqua. Commodo irure deserunt tempor nisi cillum elit nulla dolore amet pariatur aliquip irure reprehenderit aute. Cillum minim cillum irure commodo cillum eu consequat et dolore sint sunt aliquip fugiat consequat. In laboris exercitation fugiat aute laboris mollit cupidatat laboris nostrud ut. Non enim aliquip anim ullamco non incididunt proident eu. Et officia tempor exercitation magna incididunt incididunt veniam reprehenderit.

Summary Magna fugiat consectetur magna adipisicing minim id elit Lorem culpa. Ut cillum adipisicing fugiat pariatur consectetur irure. Sit voluptate mollit nulla culpa incididunt minim velit non ex nostrud ad. Amet eiusmod magna voluptate nulla exercitation sit id. Nostrud in do id exercitation excepteur minim reprehenderit anim excepteur veniam eiusmod duis proident. Eiusmod aliquip laborum deserunt officia cillum Lorem excepteur Lorem ad.

3.1.3 Hash Table

A `Hash Table` is a data structure that uses a hash function to map keys to their corresponding values. It is an efficient way to implement an associative array, where keys are used to look up values.

The basic idea behind a `Hash Table` is to use a hash function to map a key to an index in an array, called a bucket, where the corresponding value can be found or stored. The process of mapping a key to an index is called `hashing`.

Each element in a `Hash Table` consists of:

1. Key: This is the value used to look up a corresponding element in the `Hash Table`.
2. Value: This is the value associated with the key that is stored in the `Hash Table`.

When a new element is added to a `Hash Table`, the key is passed through a `hash` function which produces an `index` (also called a hash value or bucket) where the element is stored. When a value is to be retrieved, the key is passed through the same hash function, and the resulting `index` is used to look up the corresponding value in the `Hash Table`.

Table 3.4: Efficiency Analysis of Hash Table Operations

Operation	Worst Case	Average Case
Search	$O(n)$	$\Theta(1)$
Insert	$O(n)$	$\Theta(1)$
Delete	$O(n)$	$\Theta(1)$

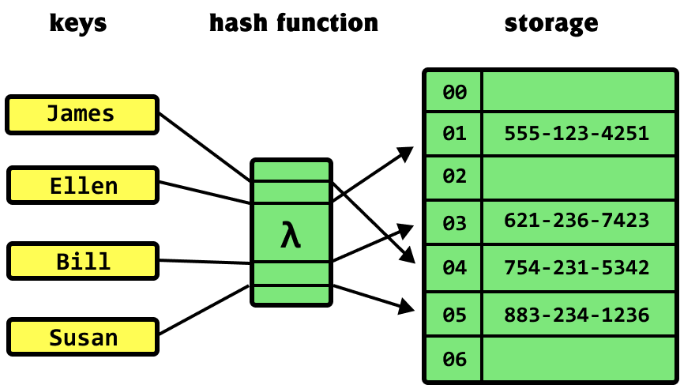


Figure 3.4: Hash Table [4]

Advantages

- Efficient searching, insertion, and deletion: A well-implemented `Hash Table` allows for these operations to be performed in `O(1)` time, which is useful for a Version Control System as it needs to be able to quickly retrieve, insert, and delete file versions.
- Dynamic resizing: `Hash Tables` can grow or shrink in size as needed, which is useful for a Version Control System as the number of file versions can vary greatly.
- Low overhead: A `Hash Table` only requires a small amount of overhead for pointers and the hash function, and thus uses less memory than an array or a `Linked List` with the same number of elements.

Disadvantages

- Hash collisions: `Hash` functions can produce collisions, where two different keys produce the same `index`, leading to the same location in the `Hash Table`.
  - `Collision resolution techniques`, such as `open addressing` and `separate chaining`, can be used to handle collisions but it still increases the time complexity of the `Hash Table` operations.

- Clustering: When all the elements in a `Hash Table` are stored in the same bucket, it is called `clustering`. This can lead to a performance degradation leading the `Hash Table` operations to have the worst-case time complexity of `O(n)` for insertion, deletion, and retrieval.

Implementation Details Non veniam nostrud occaecat magna minim officia fugiat ad tempor nulla sunt laboris deserunt minim. Officia commodo ut anim culpa dolore eiusmod ex do tempor tempor Lorem dolore aute culpa. Fugiat aliqua tempor minim esse consequat cupidatat sint dolor mollit irure consectetur fugiat. Ipsum excepteur eu cillum ea sit cillum sunt excepteur exercitation fugiat Lorem ut fugiat ut. Nulla aliquip cillum pariatur dolore aute pariatur do. Excepteur labore veniam eiusmod dolore dolor mollit eu.

Ea aliqua incididunt consectetur id officia. Dolor enim commodo pariatur enim veniam anim. Culpa laboris eu aute anim aliquip. Voluptate laboris et minim labore exercitation aliqua.

Quis voluptate duis dolore ea proident duis et ad. Proident sunt irure sint minim veniam ut laboris culpa laboris eiusmod id. Duis proident id nulla dolor est dolore. Mollit exercitation ad eiusmod ad eiusmod duis occaecat fugiat elit occaecat eu. Incididunt nisi velit quis mollit dolore cupidatat qui fugiat labore in. Est do qui enim in adipisicing eiusmod pariatur enim sit exercitation dolore officia.

Summary Minim non consequat culpa eu officia aliquip officia officia veniam in. Velit ut Lorem ea do adipisicing velit aliquip laborum. Proident dolore pariatur aliquip excepteur consectetur culpa adipisicing enim irure tempor eiusmod. Deserunt minim aliquip occaecat cillum mollit. In nulla fugiat consectetur nisi aliquip incididunt proident excepteur ex excepteur voluptate laboris. Reprehenderit amet anim incididunt aliqua est minim labore ipsum duis consectetur amet. Do mollit quis eiusmod ad consectetur ea nulla dolore voluptate nisi ea laborum est.

3.1.4 Directed Acyclic Graph (DAG)

A `Directed Acyclic Graph (DAG)` is a type of graph that consists of a set of nodes and directed edges between pairs of nodes. The edges have direction and they connect one node to another. Unlike in a `Tree` structure, in a `DAG`, a node can have multiple parents and multiple children, but there cannot be any cycles in the graph.

A node in a `DAG` can represent any type of data, and the edges can represent any type of relationship between the nodes. Each node in a `DAG` contains the following elements:

1. Data: The data that the node represents.
2. Adjacency list: A list of the nodes that are connected to the current node by an edge.

Table 3.5: Efficiency Analysis of Directed Acyclic Graph (DAG) Operations

Operation	Worst Case	Average Case

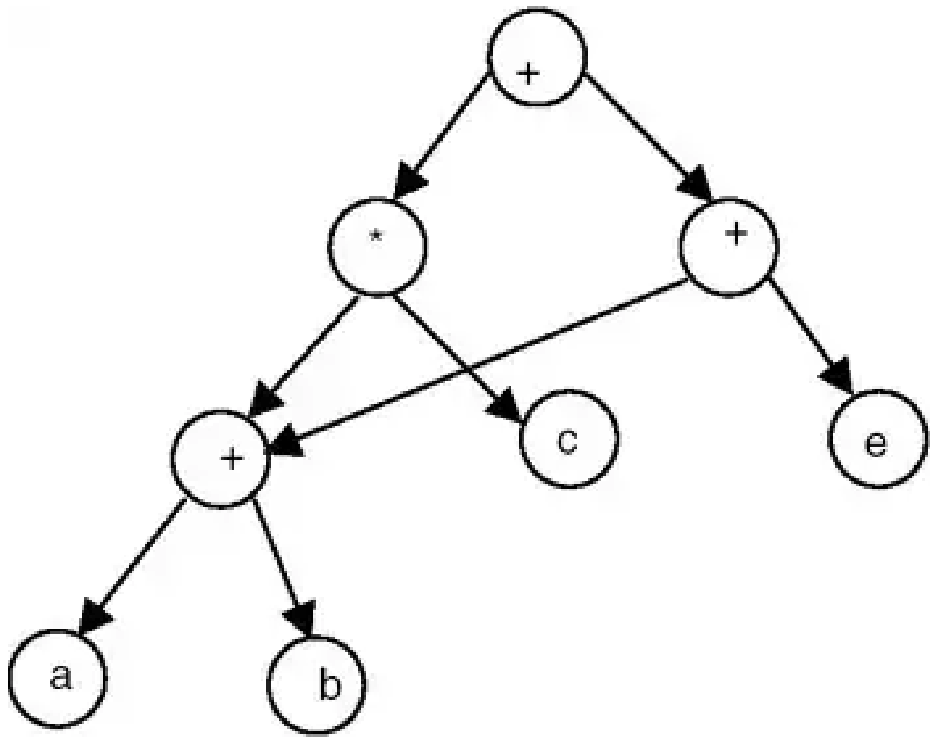


Figure 3.5: Directed Acyclic Graph (DAG) [6]

Advantages

- Representing complex relationships: `DAGs` can be useful for representing complex relationships between different versions of a file, such as `branching` and `merging` of changes.
- Representing multiple paths: `DAGs` can be used to represent multiple paths or multiple possibilities of how a file can change over time.
- Flexibility: `DAGs` are very flexible and can be used to represent any type of data and any type of relationship between the data.

Disadvantages

- Complex traversal: Traversing a `DAG` can be more complex than traversing a `Tree` because there can be multiple paths to traverse, which makes it more difficult to retrieve specific versions of a file.
- Limited scalability: `DAGs` may not scale well for large file histories and a large number of versions because they can become very complex and difficult to traverse.
- Not good for searching, insertion, and deletion: `DAGs` are not good for searching, insertion, and deletion because they are not ordered and they do not have a root node. This means that the time complexity of these operations is `O(n)` due to the need to traverse the entire graph.

Implementation Details Qui ea velit dolor cupidatat esse anim est labore. Reprehenderit veniam quis magna Lorem ad id exercitation mollit pariatur culpa officia. Dolor commodo aute eu quis Lorem ea nulla eu reprehenderit elit proident. Aliquip nostrud consequat laboris velit anim id dolor fugiat. Aute ea cupidatat velit sunt mollit id proident nostrud incididunt dolore incididunt aliquip adipisicing. Cillum et labore anim est incididunt laborum dolore commodo ullamco mollit ut ut ullamco.

Est veniam aliquip non mollit. Duis aliqua pariatur in officia cillum labore excepteur aliqua nisi. Tempor adipisicing commodo consequat occaecat ad mollit ad fugiat aliquip consectetur aliquip fugiat et. Sunt voluptate exercitation sunt esse.

Ex pariatur labore mollit labore amet. Occaecat culpa commodo ad cillum occaecat voluptate Lorem officia. Deserunt ea velit consequat sunt ex velit cillum do. Dolor nostrud in veniam laboris pariatur fugiat Lorem. Est quis sint anim deserunt elit fugiat culpa occaecat est in anim sunt ipsum.

Summary Minim ullamco duis officia fugiat anim ad in ipsum adipisicing nostrud mollit exercitation duis cupidatat. Esse occaecat enim anim aliqua qui fugiat mollit deserunt. Lorem culpa laborum ut veniam culpa in nulla et irure. Tempor nostrud magna qui sunt magna aliquip nisi qui. Magna excepteur pariatur mollit sint. Minim non officia aliquip qui laboris sunt. Exercitation aute adipisicing occaecat duis laboris.

3.2 Algorithms

3.2.1 Traversal

Depth First Search (DFS)

Advantages

Disadvantages

Implementation Details

Summary

Breadth First Search (BFS)

Advantages

Disadvantages

Implementation Details

Summary

3.2.2 Hashing

SHA-2

Advantages

Disadvantages

Implementation Details

Summary

Rabin-Karp Algorithm

Advantages

Disadvantages

Implementation Details

Summary

3.2.3 Diffing

Longest Common Subsequence (LCS)

Advantages

Disadvantages

Implementation Details

Summary

Myers' Diff Algorithm

Advantages

Disadvantages

Implementation Details

Summary

3.2.4 Merging

3-Way Merge

Advantages

Disadvantages

Implementation Details

Summary

Recursive Merge Algorithm

Advantages

Disadvantages

Implementation Details

Chapter 4

Implementation

Chapter 5

Evaluation

Chapter 6

Conclusion

List of Figures

List of Tables

Bibliography

[1] J. Cloer. "10 years of git: An interview with git creator linus torvalds." (Aug. 2019), [Online]. Available: https://www.linux.com/news/10-years-git-interview-git-creator-linus-torvalds/.

[2] C. McMahon. "Understanding binary search trees." (Jul. 2020), [Online]. Available: https://dev.to/christinamcmahon/understanding-binary-search-trees-4d90.

[3] A. S. Ravikiran. "Linked list in a data structure: All you need to know." (Nov. 2022), [Online]. Available: https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure.

[4] K. Stemmler. "Hash tables: What, why, how to use them." (Jan. 2022), [Online]. Available: https://khalilstemmler.com/blogs/data-structures-algorithms/hash-tables/.

[5] J. Stopak. "Version control systems: A technical guide to vcs internals." (Nov. 2019), [Online]. Available: https://initialcommit.com/blog/Technical-Guide-VCS-Internals.

[6] H. Surti. "Advanced data structures part 1: Directed acyclic graph (dag)." (Apr. 2016), [Online]. Available: https://medium.com/@hamzasurti/advanced-data-structures-part-1-directed-acyclic-graph-dag-c1d1145b5e5a.

[7] W. F. Tichy, "Rcs - a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985. DOI: 10.1002/spe.4380150703.

[8] R. Vaghani. "Difference between singly linked list and doubly linked list." (Jan. 2023), [Online]. Available: https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/.