Git [continued]

Cloning a repo in Git

- Is cloning same as copy?
- When you clone a repo, by default all the files get tracked by Git.
  - All the files by default in a unmodified state.
- All other files in the directory[which were not part of the last snapshot are untracked files].

Git commands

- $git config
- $git config –list
- $git config user.name
- $git clone <url>
- $git –help
- $git status

Status of the repo

- The git status command is used to check the status of the files in the current directory:

```
sunil@LAPTOP-8IVAD4JQ MINGW64 ~/git/JavaRepoDemo (feature)
$ git status
On branch feature
Your branch is up to date with 'upstream/main'.
```

- This implies that:
  - You have a clean working directory
  - No untracked files exist
  - No tracked file are in modified stage
  - No tracked file are staged
  - The current branch is origin/master

Adding untracked files

- $git add <filepath/directorypath>…
  - The mentioned files/directories get tracked.

```
sunil@LAPTOP-8IVAD4JQ MINGW64 ~/git/JavaRepoDemo (feature)
$ git status
On branch feature
Your branch is up to date with 'upstream/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newread.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .project
```

Tracked and untracked Files

- Every file in your working directory can be in *tracked* or *untracked* state.
- Tracked files are those which Git knows about;
- They were a part of the last snapshot of Git;
- They can be in any stage – unmodified, modified or staged.
- On cloning a repository, all of its files will be tracked and in unmodified state because Git has just checked them out and no modifications have been made on them so far.

- Everything else in the directory are untracked files — these files were not in the last snapshot and are not in the staging area as well.

Do the below on Git

- Adding untracked files using: git add <file>
- Check the status before and after adding.
- Modify an existing file then stage it using: git add
- Use git commit to commit the changes.
  - Eg: git commit –m “message to commit”
- git add is a multipurpose command
  - To start tracking new files
  - To stage files

Push changes to remote repo

- Using: git push origin main/master
- Note:
  - This command will work only if you have write access on the repository to which you are trying to push the changes to and if nobody has pushed in the meantime.
  - If someone else also cloned the repository at the same time and he/she pushed the changes upstream and then you try to push your changes upstream, your push will be rejected.
  - You will be required to fetch the changes made by them first, and incorporate it into yours, and then you will be able to push your changes.

Needs for Branching

- Helps to work in isolation
- It can orchestrate parallel development allowing developers to work on tasks simultaneously as part of a team
- And parallel builds and testing ensure developers get the feedback they need quickly
- Parallel tasks possible by creating multiple branches
  - While you wait for changes from one branch to be pulled by the owner, you may start your work on another branch

Branching in Git

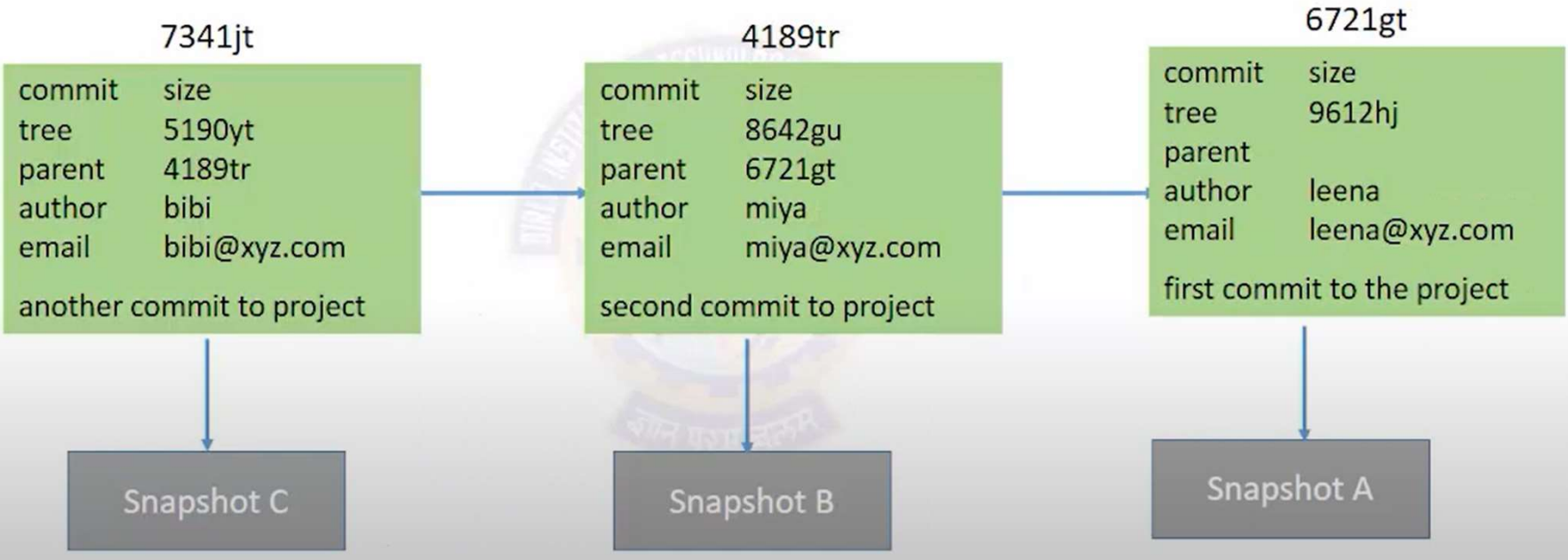
- Since Git stores data as a series of snapshots and not as a series of change sets – it is able to provide support for branching more efficiently as compared to other VCS

When a commit is done,

- A pointer to the snapshot of the staged content is stored
- This pointer object also contains author's name and email address, commit message, and pointer to the commit(s) immediately before the current commit
  - Zero parents for the initial commit
  - One parent for a normal commit
  - Multiple parents for a commit resulting from a merge of two or more branches

Branching in Git[Continued]

- Every commit stores pointer to the previous commit.



Branching in Git[Continued]

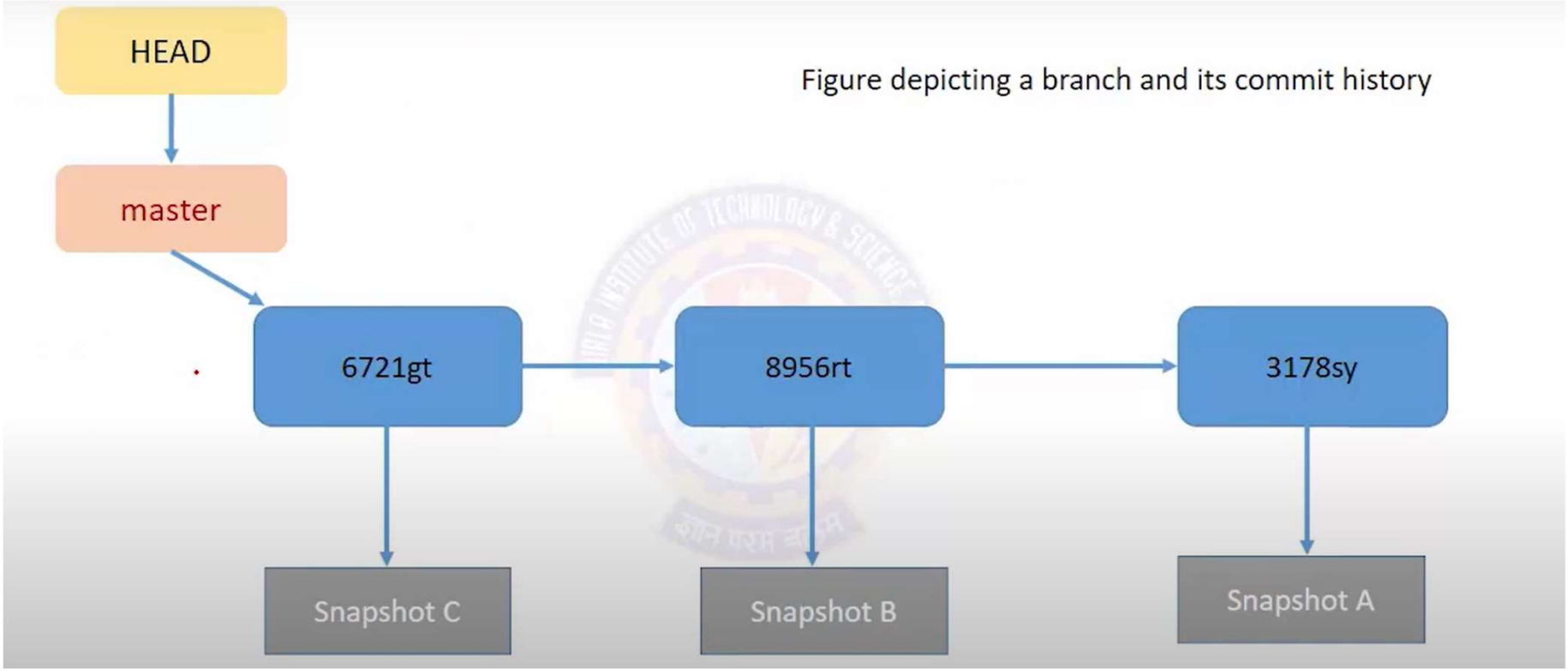


Figure depicting a branch and its commit history

Creating a new branch

- git branch <branch_name>
  - This internally creates a new pointer to the current/latest commit object.

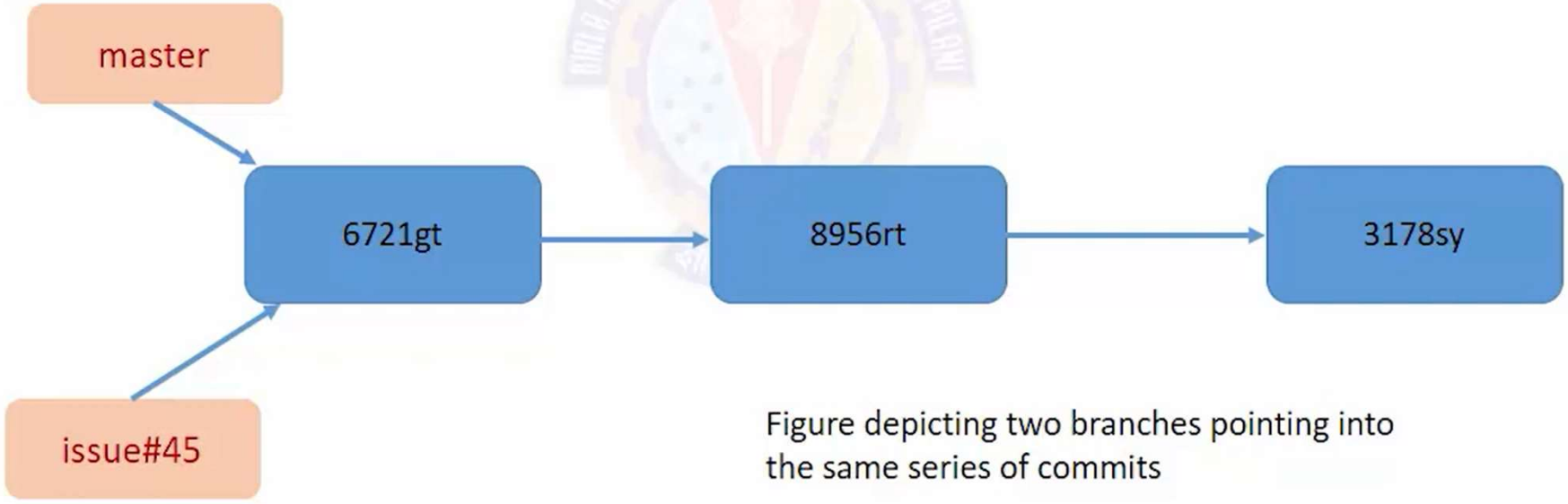


Figure depicting two branches pointing into the same series of commits

Switching to the new branch

- Git keeps a HEAD pointer to point to the current branch.
  - When a new branch is created HEAD does not automatically point to the new branch.
  - Use: git checkout <branch_name>

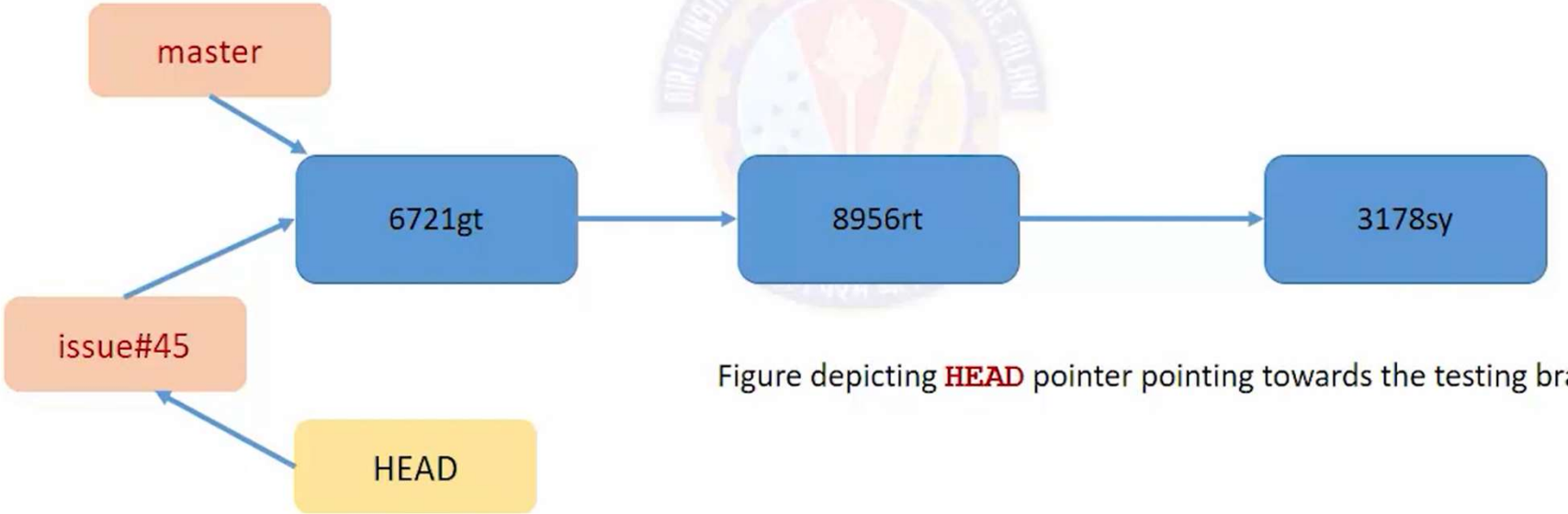


Figure depicting **HEAD** pointer pointing towards the testing branch

Making changes to the new branch

```
$ vim newTest.md
$ git commit -a -m 'added newTest.md file'
```

Because of this new change, the issue#45 branch will move ahead, while the master still points to the commit on which the check out was made

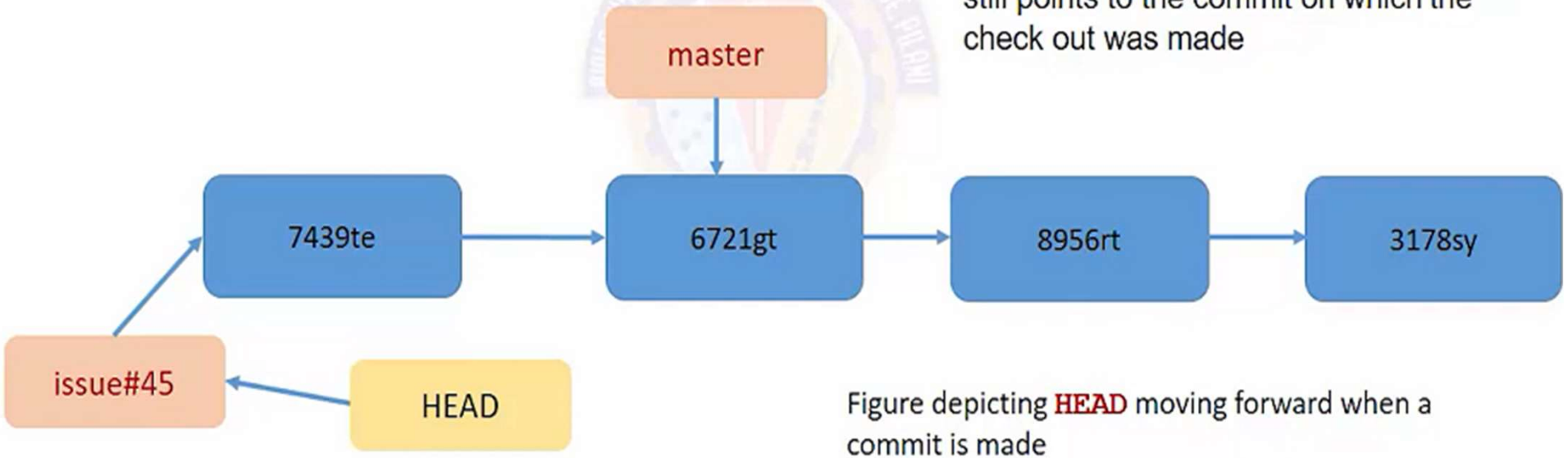


Figure depicting **HEAD** moving forward when a commit is made

- Now if you do: git checkout master
  - The HEAD pointer has moved back to point to the master
  - The files in your working directory have reverted back to a previous snapshot

```
$ git checkout master
```

- The next set of changes will now grow out of this branch

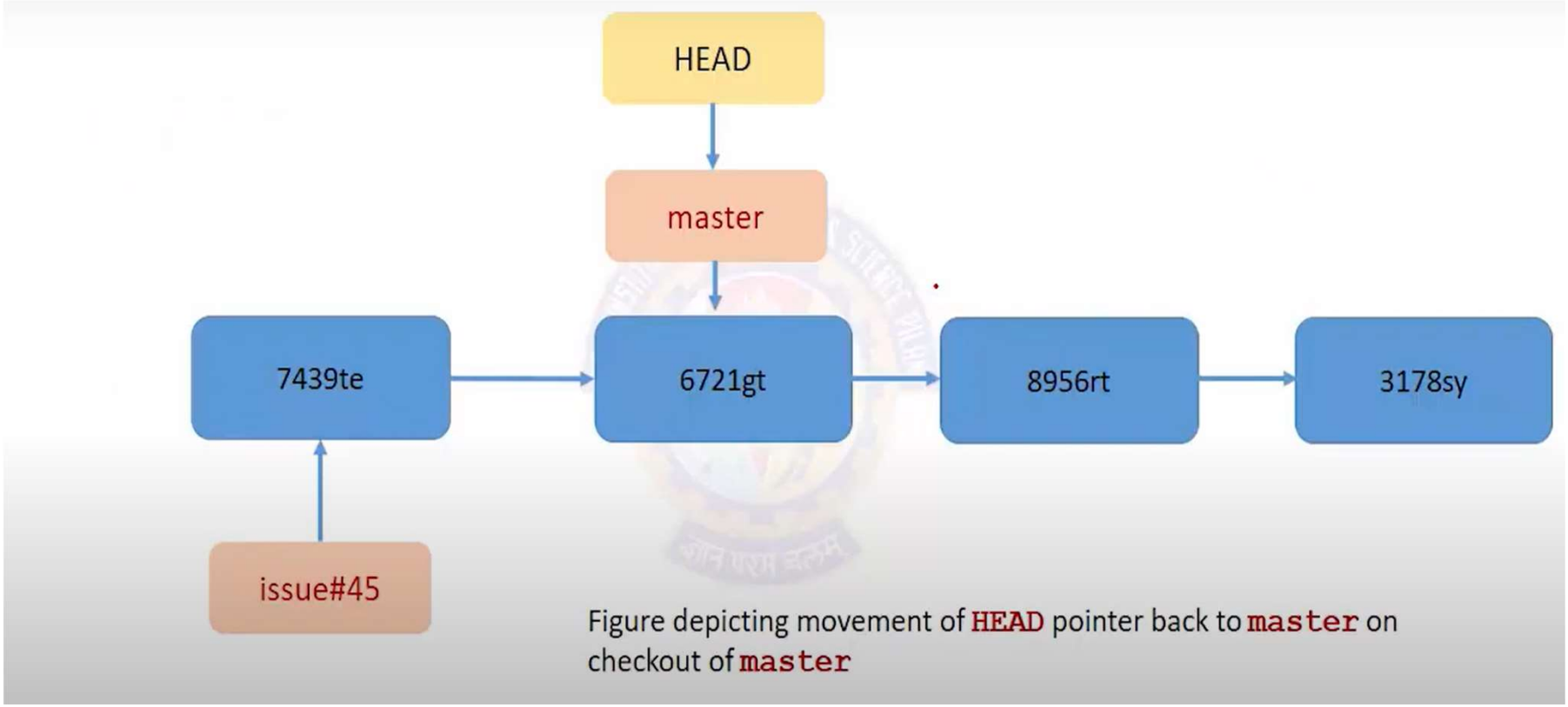


Figure depicting movement of `HEAD` pointer back to `master` on checkout of `master`

Divergent History

- Lets us do some changes to master branch.

  - Let's now make some more changes to the current `master` branch.
  - Use the following command to add another file and commit.

```
$ vim newTest2.md
$ git commit -a -m 'added newTest2.md file'
```

  - This will now create a divergent history for your project.

Divergent History [continued]

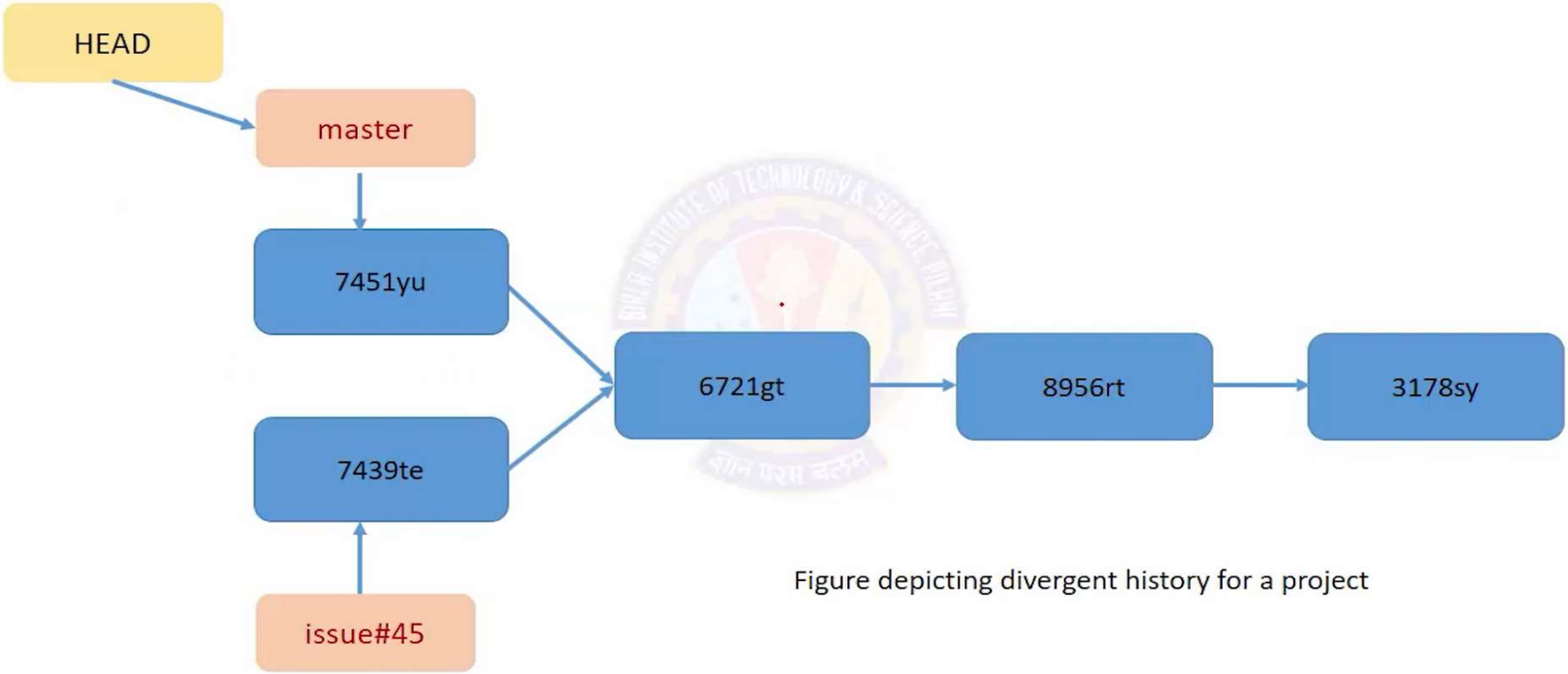


Figure depicting divergent history for a project

Merging the branch into master/main

- In order to merge the `issue#45` branch into the `master` branch, check out the branch into which the merging has to be done and then run the `git merge` command:

```
$ git checkout master
Switched to branch 'master'

$ git merge issue#45
Merge made by the 'recursive' strategy.
abc.java | 1 +
1 file changed, 1 insertion(+)
```

- In this case, Git creates a new snapshot that is created as a result of 3-way merging and also, commits automatically

After merging

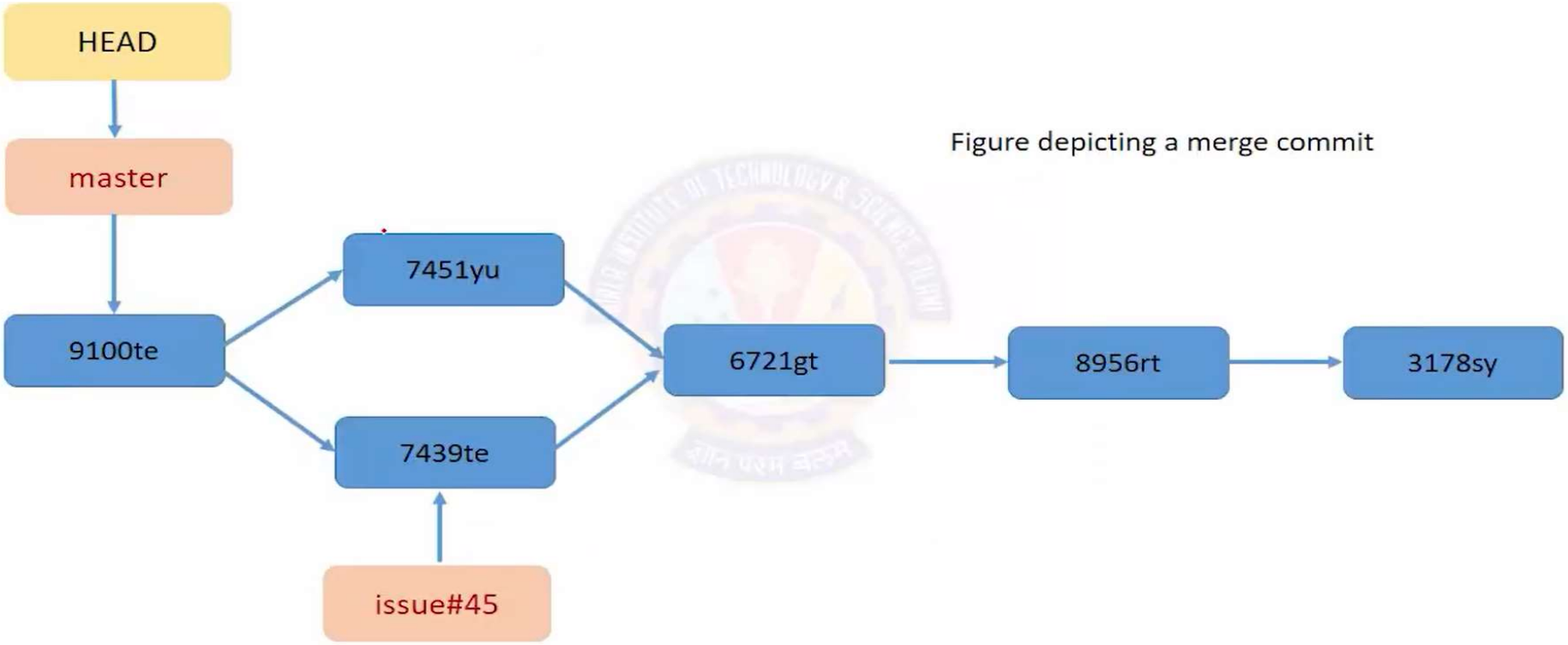


Figure depicting a merge commit

Summary

- Since Git stores data as a series of snapshots and not as a series of change sets – it is able to provide support for branching more efficiently as compared to other VCS

When a commit is done,

- A pointer to the snapshot of the staged content is stored
- This pointer object also contains author's name and email address, commit message, and pointer to the commit(s) immediately before the current commit
  - Zero parents for the initial commit
  - One parent for a normal commit
  - Multiple parents for a commit resulting from a merge of two or more branches

Merge Conflicts

Scenario 1:

- In case two or more collaborators have worked on the same file, differently in two branches that are being merged, this will NOT raise a merge conflict.
- Since there are no overlapping portions, automatic merging will be carried.

Scenario 2:

- In case two or more collaborators have worked on the same part of a file, differently in two branches that are being merged, this will raise a merge conflict.
- This is so because there are overlapping area which have been changed by the two collaborators.

Example of a conflict

- In scenario 2, running `git merge` command will result in an error being thrown, as shown below:

```
$ git merge issue#45
Auto-merging abc.java
CONFLICT (content): Merge conflict in abc.java
Automatic merge failed; fix conflicts and then commit the result.
```

- The automatic merge commit process has been paused.
- Use the `git status` command to visualize the details of the merge conflict.

```
$ git status
On branch master
You have unmerged paths.
Unmerged paths:
(use "git add <file>..." to mark resolution)
both modified: abc.java
no changes added to commit (use "git add" and/or "git commit -a")
```

- Git adds standard conflict-resolution markers to the files with conflicts.
- These markers can be visualised by opening the files and resolving them manually.

```
<<<<<<< HEAD:abc.java
System.out.println("The area is: " + area);
= = = = = = = =
System.out.println("The area of the triangle is: " + area);
>>>>>>> issue#45:abc.java
```

- The top portion of the message (above ======) represents the content of the file in the current branch (the one that you are trying to merge into).
- Markers like <<<<<< and >>>>>> are added to represent the contents of the files in the current branch and other branch.
- After resolving the changes in each of the conflicted file, run git add on each of the file to mark it as resolved. Moving the file to staging area will mark it as resolved.
- Use the mergetool or opendiff tool to resolve the conflict manually using visual interface.

Deleting a branch

- Using: git branch –d <branch_name>