CS 35L
Discussion 1A
Week 8

Git -- More

Contents

1.0 Git Internal

- What happens after running `git init`?

1.0 Git Internal

- What happens after running `git init`?
  - Initialize an empty git repo in current directory;
  - Everything inside this repo(directory) will be tracked
  - A new folder `.git` : contain version control information

1.0 Git Internal

- What happens after running `git init`?
  - Initialize an empty git repo in current directory;
  - Everything inside this repo(directory) will be tracked
  - A new folder `.git` : contain version control information
- Let's take a further look into this `.git` folder
  - Core parts:
    - **`HEAD`** (file): points to the branch you currently check out
    - **`index`** (file): where Git stores your staging area information
    - **`objects`** (dir): stores all the content for your database
    - **`refs`** (dir): stores pointers into commit objects (branches, tags, remotes and more)
  - Other parts:
    - **`description`** (file): used only by the GitWeb program, so don't worry about it.
    - **`config`** (file): contains your project-specific configuration options
    - **`info`** (dir): keeps a global exclude file for ignored patterns that you don't want to track in a .gitignore file
    - **`hooks`** (dir): contains your client- or server-side hook scripts (https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#_git_hooks)

1.0 Git Internal

```
                                    git branch
  dev
  globalGrid_partitioner
  master
  partitioner_updates
  sparse_array
* sparseindexspace_update
(END)
```

```
❯ git branch
❯ cat HEAD
ref: refs/heads/sparseindexspace_update
~/Dev/SPSIM/Cabana/.git sparseindexspace_update !1 ?3 ❯
```

- What happens after running **`git init`**?
  - Initialize an empty git repo in current directory;
  - Everything inside this repo(directory) w
  - A new folder **`.git`** : contain version con
- Let's take a further look into this **`.git`** folder
  - Core parts:
    - **`HEAD`** (file): points to the branch you currently check out
    - **`index`** (file): where Git stores your staging area information
    - **`objects`** (dir): stores all the content for your database
    - **`refs`** (dir): stores pointers into commit objects (branches, tags, remotes and more)
  - Other parts:
    - **`description`** (file): used only by the GitWeb program, so don't worry about it.
    - **`config`** (file): contains your project-specific configuration options
    - **`info`** (dir): keeps a global exclude file for ignored patterns that you don't want to track in a .gitignore file
    - **`hooks`** (dir): contains your client- or server-side hook scripts (https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#_git_hooks)

1.0 Git Internal

- What happens after running `git init`?
  - Initialize an empty git repo in current directory;
  - Everything inside this repo(directory) will be tracked
  - A new folder `.git` : contain version control information
- Let's take a further look into this `.git` folder
  - Core parts:
    - **HEAD** (file): points to the branch you currently check out
    - **index** (file): where Git stores your staging area information
    - **objects** (dir): stores all the content for your database
    - **refs** (dir): stores pointers into commit objects (branches, tags, remo
  - Other parts:
    - **description** (file): used only by the GitWeb program, so don't wor
    - **config** (file): contains your project-specific configuration options
    - **info** (dir): keeps a global exclude file for ignored patterns that you don't want to track in a .gitignore file
    - **hooks** (dir): contains your client- or server-side hook scripts (https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#_git_hooks)

1.0 Git Internal

```
git branch
  dev
  globalGrid_partitioner
  master
  partitioner_updates
  sparse_array
* sparseindexspace_update
(END)
```

- What happens after running `git init`?
  - Initialize an empty git repo in current directory;
  - Everything inside this repo(directory) will be tracked
  - A new folder `.git` : contain version control information
- Let's take a further look into this `.git` fol

```
❯ ls objects
0a  17  2b  33  3b  49  54  6b  74  84  8c  a9  b3  b9  bd  c3  cb  cf  d8  df  e7  ed  f7  ff
13  18  2e  35  45  51  5f  70  7f  86  9a  ab  b4  ba  c1  c4  cc  d5  db  e3  e8  f1  fb  info
15  24  31  37  46  52  61  72  83  8a  9b  ae  b7  bb  c2  c9  ce  d6  dd  e5  e9  f4  fd  pack
```

  - Core parts:
    - **HEAD** (file): points to the branch you currently check out
    - **index** (file): where Git stores your staging area information
    - **objects** (dir): stores all the content for your database
    - **refs** (dir): stores pointers into commit objects (branches, tags, remotes and more)
  - Other parts:
    - **description** (file): used only by the GitWeb program, so don't worry about it.
    - **config** (file): contains your project-specific configuration options
    - **info** (dir): keeps a global exclude file for ignored patterns that you don't want to track in a .gitignore file
    - **hooks** (dir): contains your client- or server-side hook scripts (https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#_git_hooks)

1.0 Git Internal

```
git branch
  dev
  globalGrid_partitioner
  master
  partitioner_updates
  sparse_array
* sparseindexspace_update
(END)
```

- What happens after running `git init`?
  - Initialize an empty git repo in current directory;
  - Everything inside this repo
  - A new folder `.git` : contain
- Let's take a further look

```
❯ ls refs
heads  remotes  tags
❯ ls refs/heads
dev  globalGrid_partitioner  master  partitioner_updates  sparse_array  sparseindexspace_update
❯ ls refs/remotes
origin  upstream
❯ ls refs/tags
0.4.0
❯ cat refs/heads/dev
b49b27e154abd5ca81fef0150e0861e17664cb36
```

  - Core parts:
    - **`HEAD`** (file): points to the
    - **`index`** (file): where Git stores your staging area information
    - **`objects`** (dir): stores all the content for your database
    - **`refs`** (dir): stores pointers into commit objects (branches, tags, remotes and more)
  - Other parts:
    - **`description`** (file): used only by the GitWeb program, so don't worry about it.
    - **`config`** (file): contains your project-specific configuration options
    - **`info`** (dir): keeps a global exclude file for ignored patterns that you don't want to track in a .gitignore file
    - **`hooks`** (dir): contains your client- or server-side hook scripts (https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks#_git_hooks)

1.1 Git Object Model

- All the history information of a project is stored in files referenced by a 40-digit “object name” like this:

```
1f3a18099a72266b46325e9a0c4faa47c1d453b5
63d49519f77488dff5a61a00a3544d5fe1c7374c
16c34ed1b23f58b28fc1819c70eced883619f3fd
```

- The name is calculated by taking the SHA1 hash of the **contents** of the object
  - SHA1: a cryptographic hash function
  - It is virtually impossible to find two different objects with the same name
- The git objects: (type, size, content)
  - size = size of the content
  - content: depend on what type of the object is
- Object types
  - `blob`: used to store file data - it is generally a file
  - `tree`: basically like a directory - it references a bunch of other trees and/or blobs
  - `commit`: points to a single tree, marking it as what the project looked like at a certain point in time
  - `tag`: a way to mark a specific commit as special
- List all objects:
  - `git rev-list --objects --all`
  - `git cat-file --batch-check='%(objectname) %(objecttype)' --batch-all-objects`
- Checkout the type of a git object
  - `git cat-file -t <SHA hash>`

```
~/Des/c/learngit  master ?1 ❯ git cat-file -t 16c34ed
commit
~/Des/c/learngit  master ?1 ❯ git cat-file -t 90fb4f10
blob
```

1.1 Git Object - blob

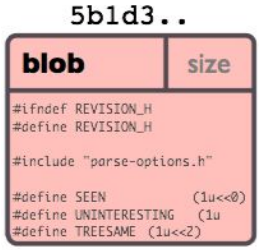
- `blob`: generally stores the contents of a file
  - A chunk of binary data
  - Doesn't refer to anything else or have attributes of any kind, not even a file name
  - The blob is entirely defined by its data (has nothing to do with the file name):
    - Same contents => same blob object
    - Same contents => same SHA hash
- Check the contents of any blob:
  - **`git show <SHA hash>`**

```
blob 90fb4f10c7f3f9380e09383f74bd3e3959b4dda8    readme
```

```
~/Des/c/learngit  master ?1 ❯ git show 90fb4f10
~/Des/c/learngit  master ?1 ❯ cat readme
Git is a distributed version control system.
Git is free software under the GPL(General Public Lisence).
Git has a mutable index called stage.
Creating a new branch is quick and easy.
Test fast-forward merge.
Test non-fast-forward merge.
```

```
git show 90fb4f10
git show 90fb4f10 111x24

Git is a distributed version control system.
Git is free software under the GPL(General Public Lisence).
Git has a mutable index called stage.
Creating a new branch is quick and easy.
Test fast-forward merge.
Test non-fast-forward merge.
(END)
```

1.1 Git Object - blob



- `blob`: generally stores the contents of a file
  - A chunk of binary data
  - Doesn't refer to anything else or have attributes of any kind, not even a file name
  - The blob is entirely defined by its data (**has nothing to do with the file name**):
    - **Same contents => same blob object**
    - **Same contents => same SHA hash**
-

```
❯ ls
testfile
❯ git commit -m "first commit"
[master (root-commit) 925fd2c] first commit
 1 file changed, 4 insertions(+)
 create mode 100644 testfile
❯ git rev-list --objects --all

925fd2cef6c1d7eb0ecb28a9da797b20280cf2a9
9d926bb2dda97b9bc2d4af658177293b6a2750e0
230ee42402690ab966327172559a760dc7fe71e9 testfile
```

```
❯ cp testfile testfile_copy
❯ git add .
```

```
❯ ls
testfile  testfile_copy
❯ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   testfile_copy

❯ git status
❯ git commit -m  "add copy"
[master e1b795c] add copy
 1 file changed, 4 insertions(+)
 create mode 100644 testfile_copy
```

1.1 Git Object - blob

- `blob`: generally stores the contents of a file
  - A chunk of binary data
  - Doesn't refer to anything else or have att
  - The blob is entirely defined by its data (**h**
    - **Same contents => same blob obje**
    - **Same contents => same SHA hash**

```
5b1d3..
❯ ls
testfile  testfile_copy
❯ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   testfile_copy

❯ git status
❯ git commit -m  "add copy"
[master e1b795c] add copy
 1 file changed, 4 insertions(+)
 create mode 100644 testfile_copy
```

-

```
❯ ls
testfile
❯ git commit -m "first commit"
[master (root-commit) 925fd2c] first commit
 1 file changed, 4 insertions(+)
 create mode 100644 testfile
❯ git rev-list --objects --all

925fd2cef6c1d7eb0ecb28a9da797b20280cf2a9
9d926bb2dda97b9bc2d4af658177293b6a2750e0
230ee42402690ab966327172559a760dc7fe71e9 testfile
```

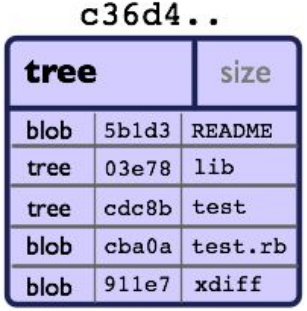
```
❯ cp testfile testfile_copy
❯ git add .
```

```
❯ git cat-file --batch-check='%(objectname) %(objecttype)' --batch-all-objects
0a4fcb855cecb952023cebb946677f0f3d6aa08c tree
230ee42402690ab966327172559a760dc7fe71e9 blob
925fd2cef6c1d7eb0ecb28a9da797b20280cf2a9 commit
9d926bb2dda97b9bc2d4af658177293b6a2750e0 tree
e1b795c9ac7614f8181f53cd8076d763f4860343 commit
```

```
❯ git cat-file 9d926bb -p
100644 blob 230ee42402690ab966327172559a760dc7fe71e9    testfile
```

```
❯ git cat-file 0a4fcb855cecb952023cebb946677f0f3d6aa08c -p
100644 blob 230ee42402690ab966327172559a760dc7fe71e9    testfile
100644 blob 230ee42402690ab966327172559a760dc7fe71e9    testfile_copy
```

1.1 Git Object - tree



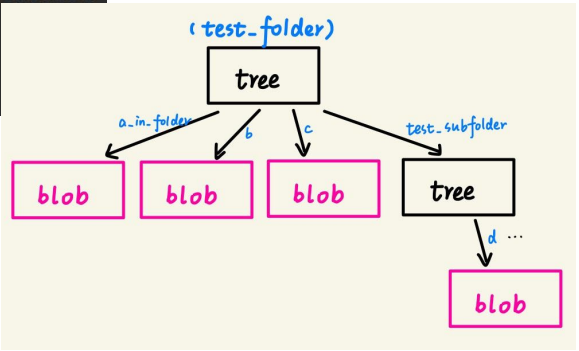
- `tree`: a simple object that has a bunch of pointers to blobs and other trees
  - Generally, it represents the contents of a directory or subdirectory
- Check the contents of any tree:
  - **`git show <SHA hash>`**

```
tree f15f9667f2afed857635e83e42cc17c0906d7525    test_folder
~/Des/c/learngit master !3 ❯ ls test_folder
a_in_folder  b_in_folder  c_in_folder  test_subfolder
~/Des/c/learngit master !3 ❯ git show f15f9667f2afed857635e83e42cc17c0906d7525
```

```
tree f15f9667f2afed857635e83e42cc17c0906d7525

a_in_folder
b_in_folder
c_in_folder
test_subfolder/
(END)
```

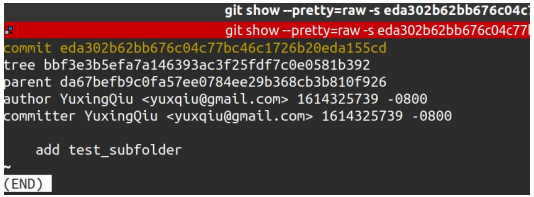
  - **`git ls-tree <SHA hash>`**

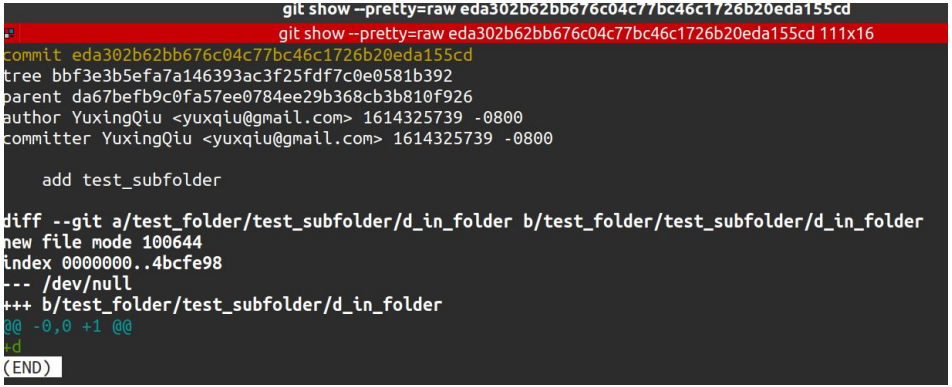
```
~/Des/c/learngit master !3 ❯ git ls-tree f15f9667f2afed857635e83e42cc17c0906d7525
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    a_in_folder
100644 blob 61780798228d17af2d34fce4cfbdf35556832472    b_in_folder
100644 blob f2ad6c76f0115a6ba5b00456a849810e7ec0af20    c_in_folder
040000 tree 723f4575a99cfed82508fa44242a7b0012c0f576    test_subfolder
```



1.1 Git Object - commit

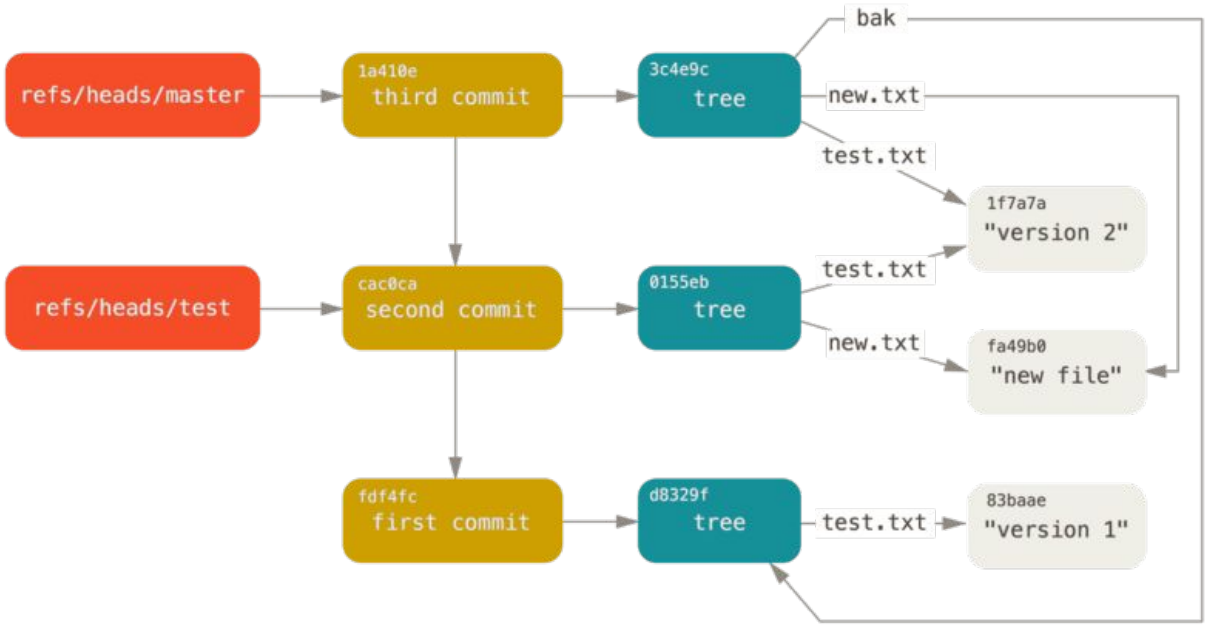
- `commit`: links a physical state of a tree with metadata
  - Description of how we got there and why
  - It contains:
    - Tree: The SHA1 name of a tree object, representing the contents of a directory at a certain point in time
    - Parent commit: The SHA1 name of some number of commits which represent the immediately previous step(s) (eg: merging commits may have multiple parent nodes)
    - Author: person responsible for this change (originally wrote the patch)+ date
    - Committer: person who actually created the commit on behalf of the author(last applied the patch) + date
    - Comment: describing this commit
  - Difference between author and commiter: https://ivan.bessarabov.com/blog/git-author-committer
- Check the contents of any commit:
  - `git log`
  - `git show --pretty=raw <SHA hash>`
  - `git show -s --pretty=raw <SHA hash>`





1.1 Git Object - Overall Picture

- `commit`: points to overall snapshot and top-level directory (`tree`)
- `tree`: contains pointers to the file and any sub-directories (`tree`)
- `blob`: contains file contents (the same file but different content/versions => different `blob`)

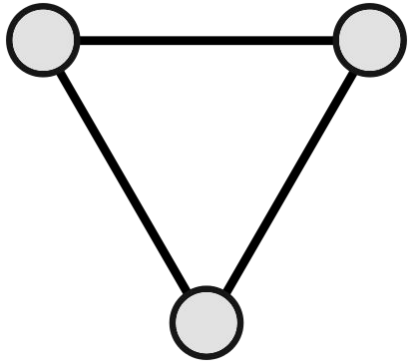


Contents

2.0 Graph Theory

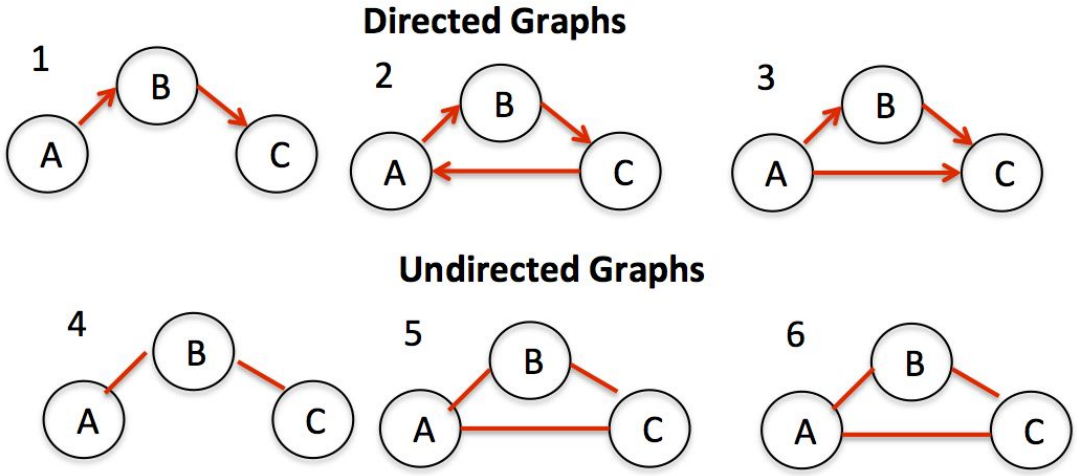
- In CS, graph is an important data structure
  - Network of communication
  - Data organization
  - Flow of computation
  - ……
- Graph
  - An ordered pair G = (V, E)
  - V: a set of vertices (nodes, points …)
  - E: a set of edges (links, lines …)

$$E \subseteq \{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$$



2.1 Directed vs Undirected Graph

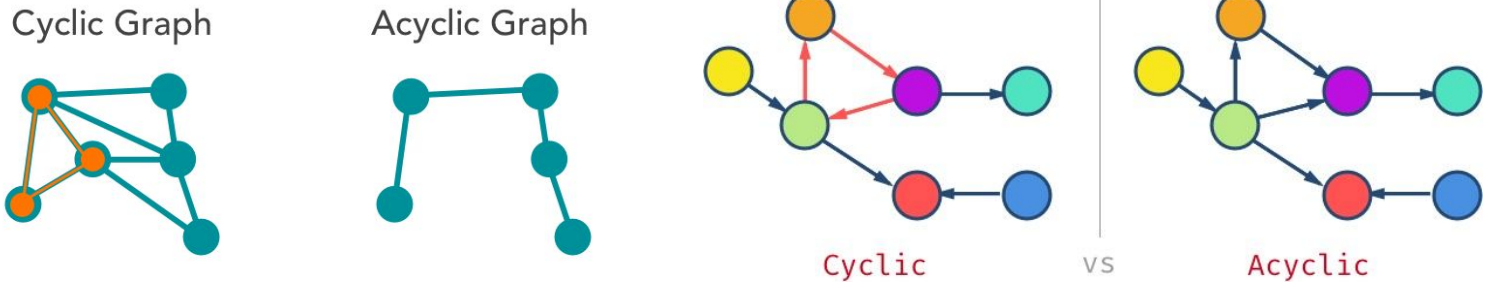
- Directed Graph: edges have orientations
- Undirected Graph: edges just show general connections, no directions



- What kind of Graph do you think git is?

2.2 Cyclic vs Acyclic Graph

- Cyclic Graph: contains a cycle or is a cycle
  - Cycle: loop; starting from one node, we can get back to itself with the paths
- Acyclic Graph: graph without any directed cycles



- What kind of Graph do you think git is?

2.3 DAG - Directed Acyclic Graph

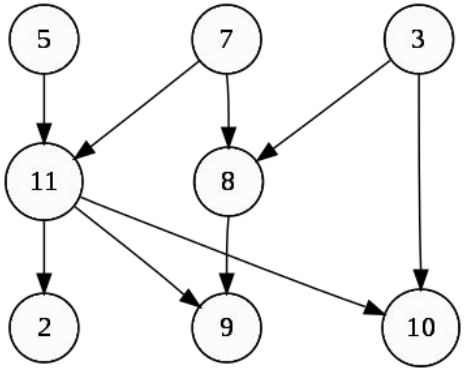
- Git is a DAG: directed, acyclic, graph
- We can process the git information with a graph data structure
- Show git log in a graph way with `git log --graph --pretty=oneline --abbrev-commit`

```
* eda302b (HEAD -> master) add test_subfolder
* da67bef add test_folder
* 53b99a1 test folder
*   c847cf9 (origin/master) merge with no-ff
|\
| * 2df88b5 (dev) test merge with no-ff
|/
* 3e27deb test fast-forward merge
*   ff484ee fix conflict
|\
| * 20fd55f (feature) add and easy
* | 4335e46 add & easy
|/
* 0f1caa8 branch dev test
* e8f3dc5 remove tmp
* 5811aa8 add test
* 30b788f understand how stage works
* 1f3a180 append GPL
* 63d4951 add distributed
* 16c34ed create a readme file
(END)
```

- In assignment 6:
  - Organize the commit info(nodes) with a DAG
  - Sort the order of commits based on their dependencies
    - Topological Sort (later)

2.4 Topological Ordering of a Directed Graph

- How to define the order in a directed graph?
  - 5, 7, 3, 11, 8, 2, 9, 10 (visual top-to-bottom, left-to-right)
  - 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
  - 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
  - 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
  - 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
  - 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)
  -



- Basic principle:
  - If we have an edge (`u` => `v`), then `u` should comes before `v` in the ordering
  - Topological ordering is possible iff. the graph has no directed cycles

Contents

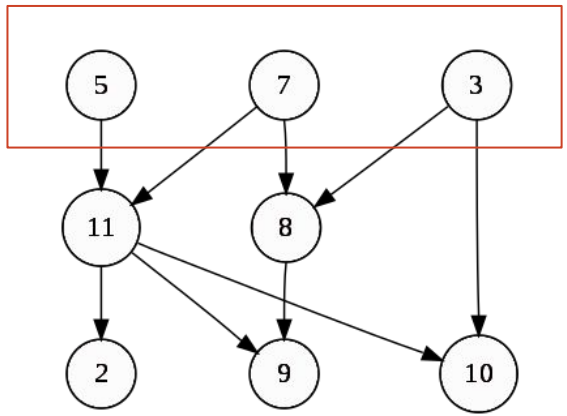
3.1 Topological Sort Algorithm

- Two typical approaches
  - Depth First Search (DFS)
  - **Khan's Algorithm**

3.2 Khan's Algorithm

- First, find a list of "start nodes" which have no incoming edges and insert them to a set *s* (at least one such node mush exist in a non-empty acyclic graph)

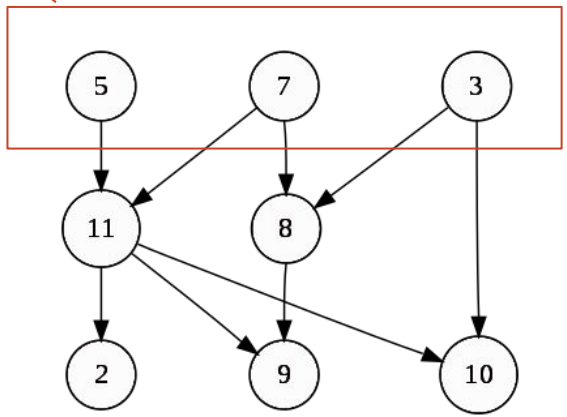
```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```



3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set *S* (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

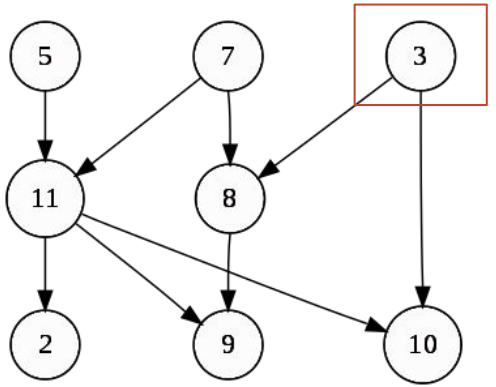


- *S* = (3, 5, 7)
- *L* = [ ]
- *E* = [(5, 11), (7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9), (3, 8), (3, 10)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set *s* (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

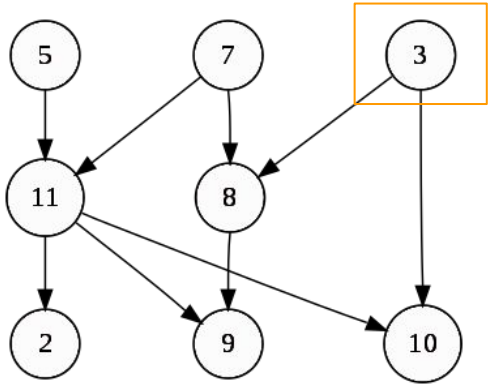


- S = (3, 5, 7) => (5, 7), n = 3
- L = [ ] => [3]
- E = [(5, 11), (7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9), (3, 8), (3, 10)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set *s* (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

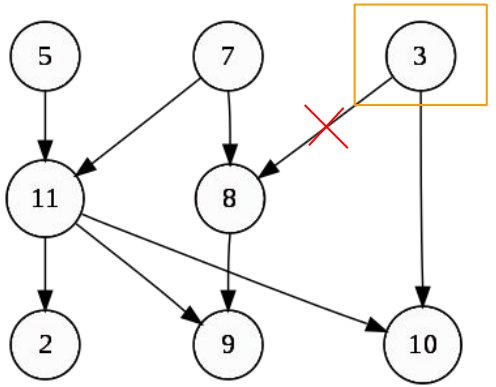


- S = (5, 7), **n = 3**
- L = [3]
- E = [(5, 11), (7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9), (3, 8), (3, 10)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
    return L    (a topologically sorted order)
```

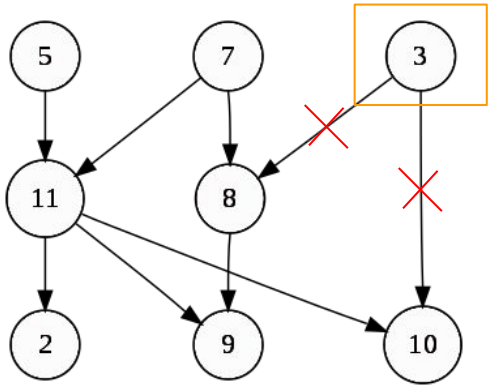


- S = (5, 7), **n = 3**
- L = [3]
- E = [(5, 11), (7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9), (3, 10)], **m = 8**

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

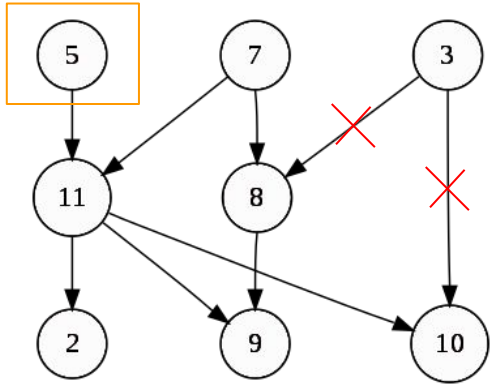


- S = (5, 7), n = 3
- L = [3]
- E = [(5, 11), (7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9)], m = 10

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

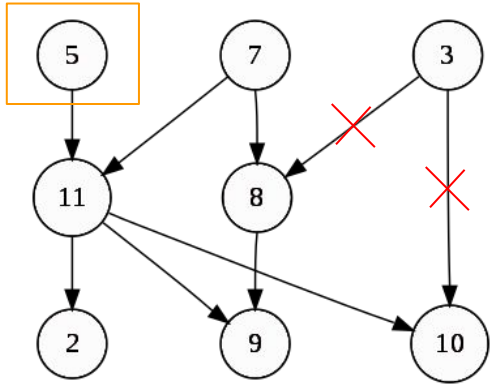


- S = (5, 7) => (7), n = 5
- L = [3] => [3, 5]
- E = [(5, 11), (7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

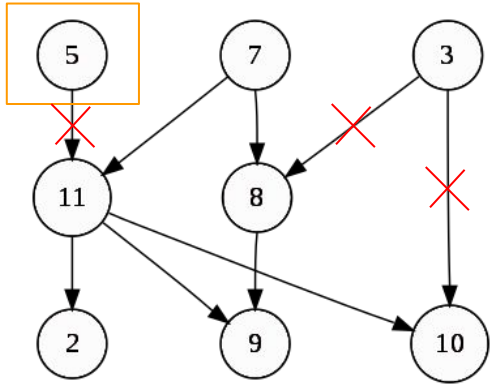


- S = (7) , **n = 5**
- L = [3, 5]
- E = [(5, 11), (7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

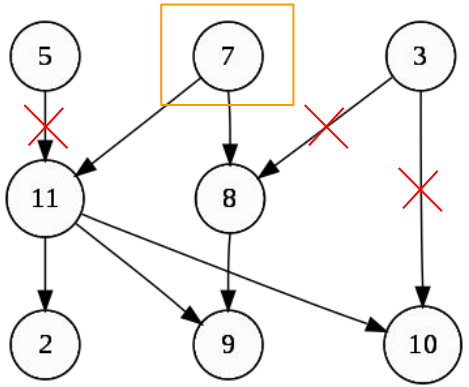


- S = (7) , **n = 5**
- L = [3, 5]
- E = [**(7, 11)**, (11, 2), (11, 9), (11,10), (7, 8), (8, 9)], **m = 11**

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

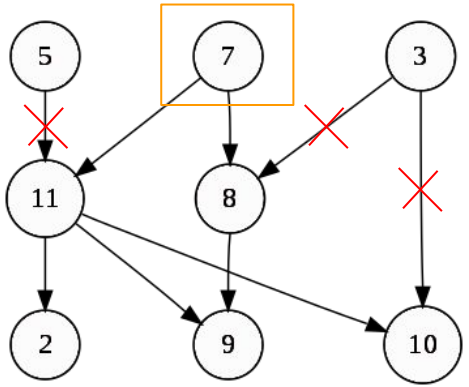


- S = (7) => ( ), n = 7
- L = [3, 5] => [3, 5, 7]
- E = [(7, 11), (11, 2), (11, 9), (11,10), (7, 8), (8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

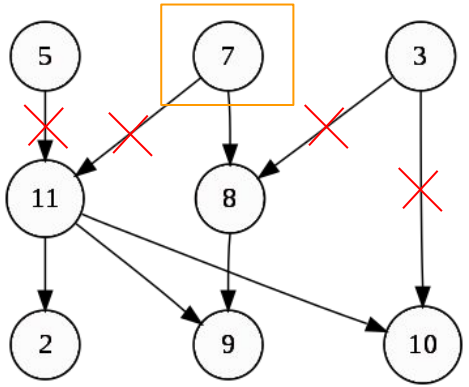


- S = ( ), **n = 7**
- L = [3, 5, 7]
- E = [(7, 11), (11, 2), (11, 9), (11,10), (7, 8) (8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

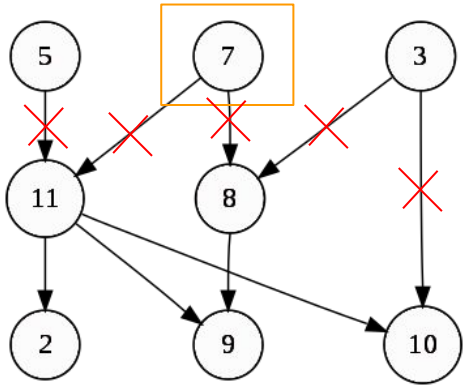


- S = ( ) => (11), n = 7
- L = [3, 5, 7]
- E = [(11, 2), (11, 9), (11,10), (7, 8), (8, 9)], m = 11

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

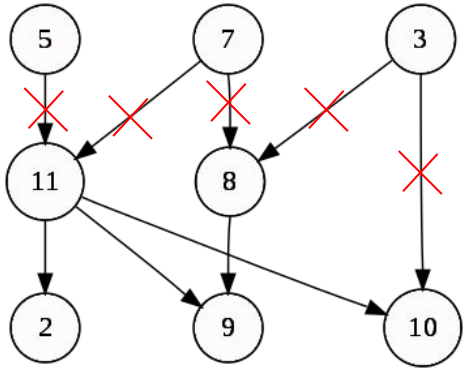


- S = (11) => (11, 8), n = 7
- L = [3, 5, 7]
- E = [(11, 2), (11, 9), (11,10), (8, 9)], m = 8

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

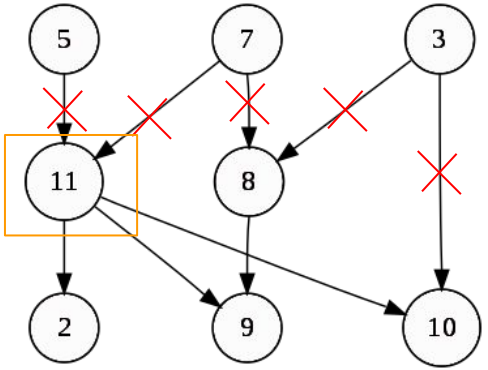


- S = (11, 8)
- L = [3, 5, 7]
- E = [(11, 2), (11, 9), (11,10), (8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

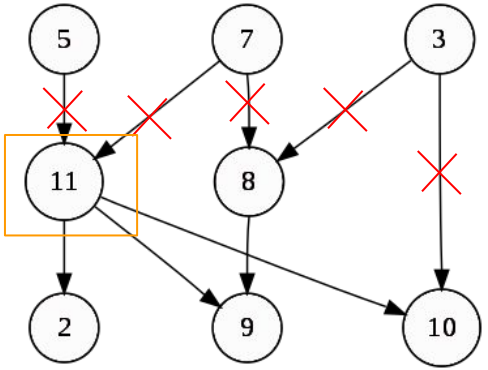


- S = (11, 8) => (8), n = 11
- L = [3, 5, 7] => [3, 5, 7, 11]
- E = [(11, 2), (11, 9), (11,10), (8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

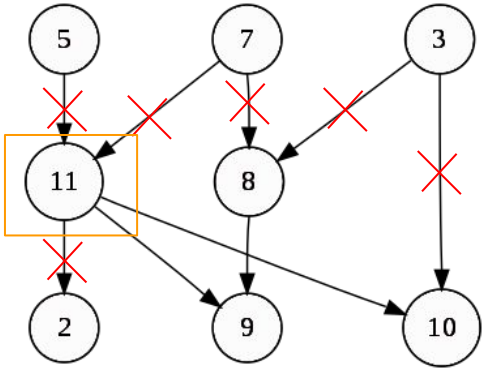


- S = (8), n = 11
- L = [3, 5, 7, 11]
- E = [(11, 2), (11, 9), (11,10), (8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

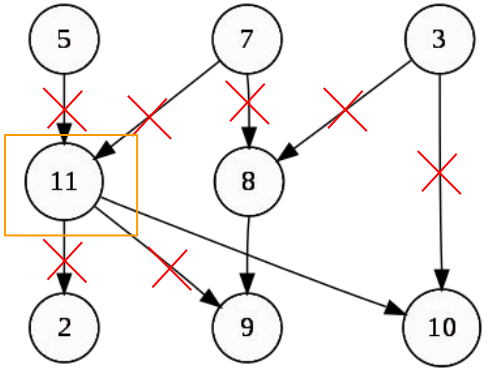


- S = (8) => (8, 2), n = 11
- L = [3, 5, 7, 11]
- E = [(11, 9), (11,10), (8, 9)], m = 2

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

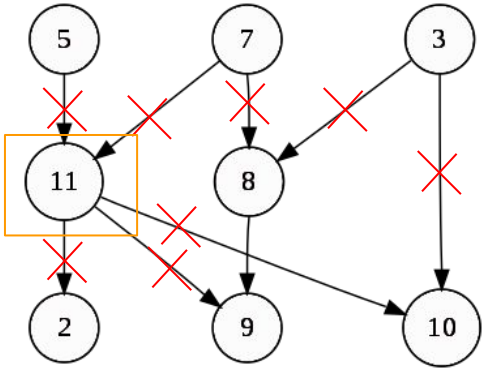


- S = (8, 2), n = 11
- L = [3, 5, 7, 11]
- E = [(11,10), (8, 9)], m = 9

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

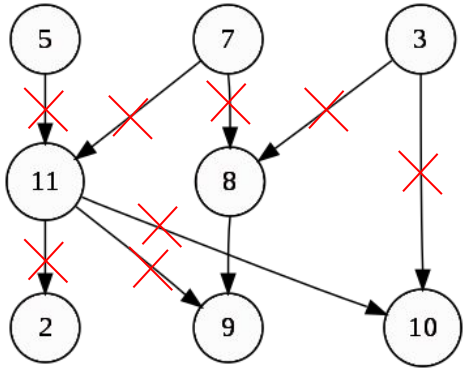


- S = (8, 2) => (8, 2, 10), n = 11
- L = [3, 5, 7, 11]
- E = [(8, 9)], m = 10

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

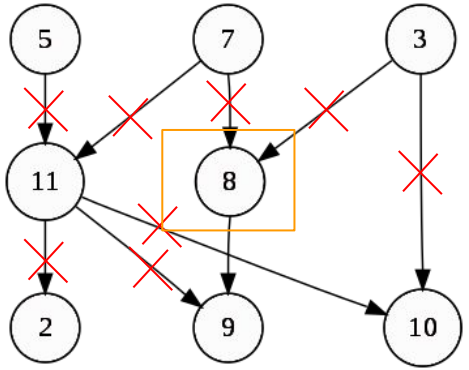


- S = (8, 2, 10)
- L = [3, 5, 7, 11]
- E = [(8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

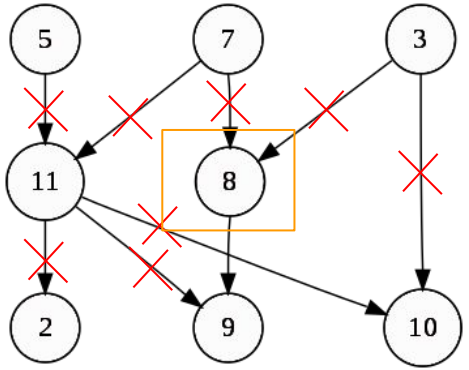


- S = (8, 2, 10) => (2, 10), n = 8
- L = [3, 5, 7, 11] => [3, 5, 7, 11, 8]
- E = [(8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

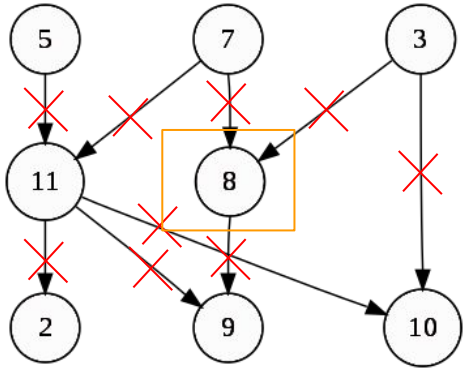


- S = (2, 10), n = 8
- L = [3, 5, 7, 11, 8]
- E = [(8, 9)]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

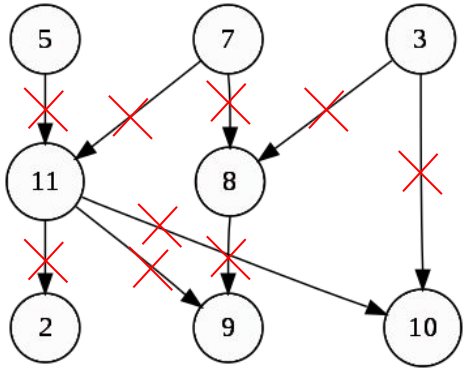


- S = (2, 10) => (2, 10, **9**), **n = 8**
- L = [3, 5, 7, 11, 8]
- E = [ ], **m = 9**

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

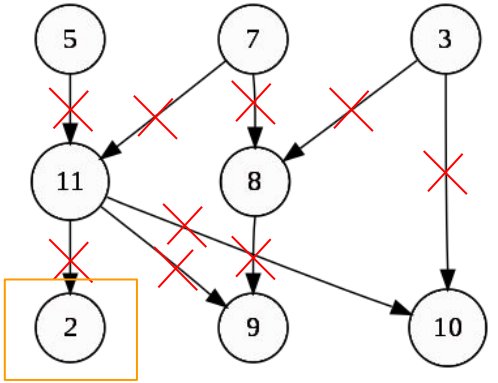


- S = (2, 10, 9)
- L = [3, 5, 7, 11, 8]
- E = [ ]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

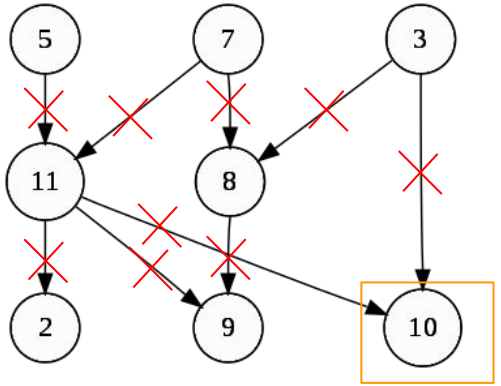


- S = (2, 10, 9) => (**10, 9**), **n = 2**
- L = [3, 5, 7, 11, 8] => [3, 5, 7, 11, 8, **2**]
- E = [ ]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

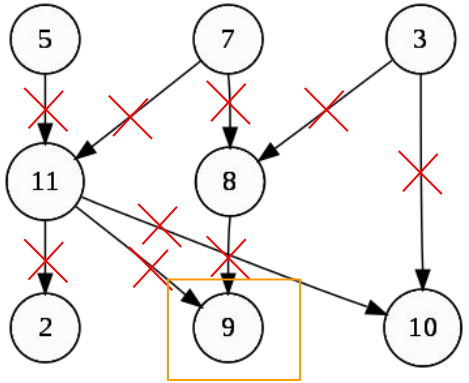


- S = (10, 9) => (9), n = 10
- L = [3, 5, 7, 11, 8, 2] => [3, 5, 7, 11, 8, 2, 10]
- E = [ ]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***s*** (at least one such node mush exist in a non-empty acyclic graph)

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

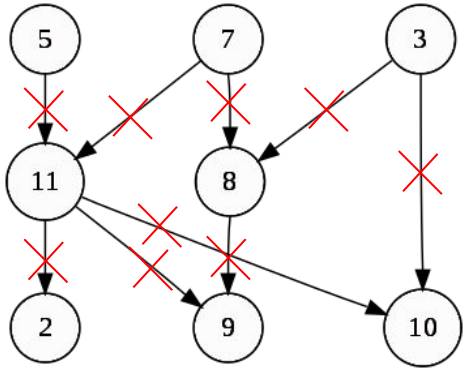


- S = (9) => ( ), **n = 9**
- L = [3, 5, 7, 11, 8, 2, 10] => [3, 5, 7, 11, 8, 2, 10, **9**]
- E = [ ]

3.2 Khan's Algorithm - Example

- First, find a list of "start nodes" which have no incoming edges and insert them to a set ***S*** (at least one such node mush exist in a non-empty acyclic graph)

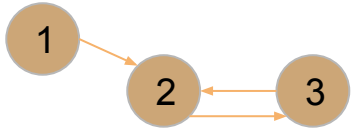
```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error    (graph has at least one cycle)
else
     return L    (a topologically sorted order)
```

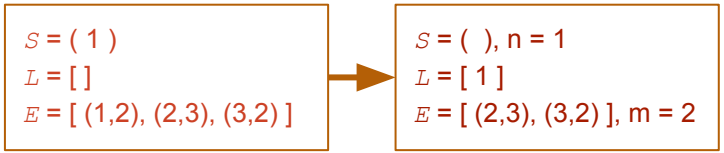


- S = ( )
- L = [3, 5, 7, 11, 8, 2, 10, 9]
- E = [ ]

3.2 Khan's Algorithm - More

- Does it matter which node we pop from the "processing" set?
  - No! Eventually we get a topological order.
- When will the algorithm fail (return error)?
  - What happens if there's a cycle?





- As long as we follow the basic principles, we are fine
- Basic principle:
  - If we have an edge (u => v), then u should comes before v in the ordering
  - Topological ordering is possible iff. the graph has no directed cycles

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge

while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S

if graph has edges then
    return error   (graph has at least one cycle)
else
    return L   (a topologically sorted order)
```