CSE 100 Project 1: Huffman Encoding and Compression

Checkpoint Deadline: **Tuesday, May. 19th, 11:59 pm**

Final Deadline: **Tuesday, May. 26th, 11:59 pm**

Overview

- In this project, you will be asked to complete two major tasks. The first is to implement a Huffman Coding Tree and use it to perform pseudo compression without using bitwise i/o. The second part is to implement bitwise i/o with arbitrary buffer size to allow true compression with efficient header. **Make sure to *read these instructions carefully*** before you start writing code or post questions on piazza.
- This is a Pair Programming project. You may ask Professors/TAs/Tutors for some guidance and help, but you can not copy code from anywhere including online sources such as Github. You may also discuss the assignment conceptually with your classmates, including bugs that you ran into and how you fixed them. However, do not look at or copy code, as this constitutes an Academic Integrity Violation. And you may not use other libraries' implementations in your solutions.
- There is a FAQ page for Docker and devcontainer setup issues please check this as well as existing Piazza posts before making your own post. Thank you!
- Here is a post on Piazza with Project 1 FAQs from previous terms
- ***Start early, submit early and often! The projects are likely to take you significantly more time than the PAs.***

Table of Contents

Pair Programming Partner Instructions

If you wish to work with a partner, make sure you carefully read the pair programming policy on the course website:https://ucsd-cse100-s20.github.io/. When submitting with a partner, make sure to submit your work on Gradescope using only ONE of the accounts and do not forget to add the other teammate's account into the submission. If you submit the checkpoint assignment with a partner, you have committed to also submitting the final assignment with a partner, so be careful in selecting a suitable partner.

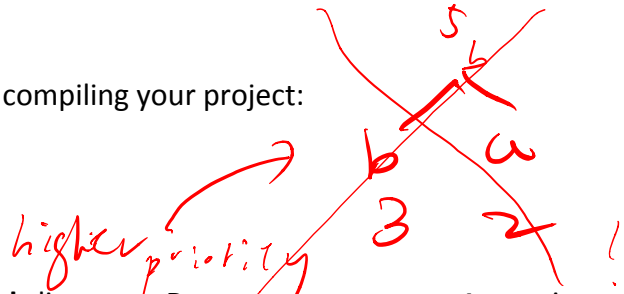
Academic Integrity and Honest Implementations

We will use automated tools to look for plagiarism or deception. **Attempting to solve a problem by other than your own means will be treated as an Academic Integrity Violation.** This includes all the issues discussed in the Academic Integrity Agreement, but in addition, it covers deceptive implementations. For example, if you use a library (create a library object and just reroute calls through that library object) rather than writing your own code, that's seriously not okay and will be treated as dishonest work.

Familiarizing Yourself with the Starter Code

Retrieve the starter code here:
https://classroom.github.com/a/KuakQGrO
Keep your github project **private**! Making it public is a violation of academic integrity.

If you are working with a partner, add them as a collaborator on your GitHub repo under "Settings" > "Manage access".



These are the 3 commands you should run to begin compiling your project:

```
mkdir build && cd build
cmake ..
make
```

⚠ Only run the cmake command inside your **build** directory. Do **not** try to run `cmake .` in your project root, because this will clutter your project with autogenerated build files.

We provided you all the `CMakeLists.txt` files except the ones in the bitstream folder, which you need to implement only for the final submission (this means for checkpoint you don't need to write any CMakeLists.txt). The `CMakeLists.txt` are a specification for the CMake tool and

they allow CMake to automatically generate the Makefiles you need to make the project (how cool is that!).

Among the starter code files, only the following files are relevant to implementation and testing. You are required to only implement the underlined files, and you must leave all the non-yellow files unchanged.

```
In src folder:
    -   FileUtils.hpp
    -   bitconverter.cpp
    -   compress.cpp
    -   uncompress.cpp
    -   CMakeLists.txt
In src/encoder folder:
    -   HCNode.hpp
    -   HCTree.hpp
    -   HCTree.cpp
    -   CMakeLists.txt
In src/bitStream/input folder:
    -   BitInputStream.hpp
    -   BitInputStream.cpp
In src/bitStream/output folder:
    -   BitOutputStream.hpp
    -   BitOutputStream.cpp
In test folder:
    -   test_BitOutputStream.cpp
    -   test_BitInputStream.cpp
    -   test_HCTree.cpp
    -   CMakeLists.txt
In data folder:
    -   check.txt
    -   bitstream_1.txt
    -   bitstream_2.txt
    -   file_2.txt
    -   file_5.txt
    -   large_urls.txt
In the top level folder:
    -   solution-bitconverter.executable
    -   solution-compress.executable
    -   solution-uncompress.executable
```

HCNode.hpp

To complete the assignment, you will modify the files in the starter code to provide full definitions for methods marked with // TODO. To see all of the TODO tags, you can run the command `grep -r TODO .` in your Project1 directory.

We personally recommend using grep like this as you can manipulate the arguments after "A" and "B" to get a good snapshot of what is required: `grep -r -A 5 -B 5 TODO *`
To search in VSCode, you can use Ctrl+Shift+F or Cmd+Shift+F on mac

You can find all the relevant information about compiling your code and running tests in the **README.md** file of the starter code.

Style Requirement

The project ships with an auto formatter based on the Google C++ Style Guide that can be executed by running "`make format`" in the build directory. VSCode is also configured to run the auto formatter on file saves. Gradescope will check that your project is properly formatted, so use the auto formatter!

Testing & Code Coverage

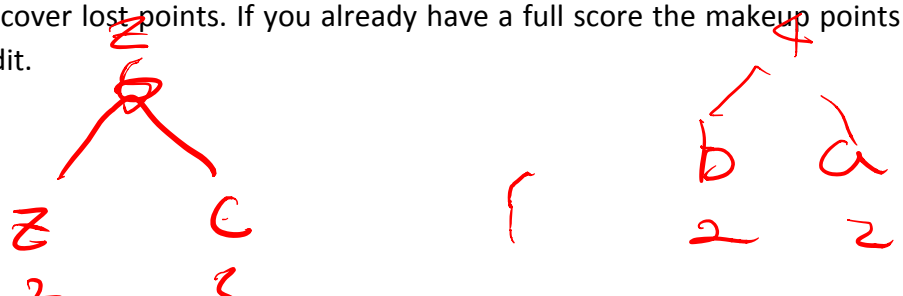
You should make good use of the provided tools, such as the GDB debugger and cppcheck. Try out the different tools in `README.md` to see how they can help you to test the program. Remember to constantly back up your code to GitHub.

Unit tests and code coverage

You will want to be sure to unit test your code at every step. You should write your own testers for every method, no matter how simple it is. Always start with simple cases in unit tests, as unit tests are extremely helpful for catching missed edge cases and any unintentional mistakes. Use larger test cases only after your program passed on small and simple test cases.

To test HCTree, it would be a good idea to use TestFixture in your tests so that multiple tests can run on the same HCTree. You can refer to the provided tests in PA1 for an example. For final submission, you should use the code that was commented out as examples to test bitstream. The inline comment should give you enough information to understand how it works.

For checkpoint submission, **the line coverage** in src/encoder directory should be at least 90% to get 10 *makeup points* for the coverage report. For final submission, the requirement becomes 90% for line coverage in both the src/encoder and src/bitStream directories. The makeup points are a chance for you to recover lost points. If you already have a full score the makeup points will *not* count as extra credit.



Submitting Your PA

Instructions to submit your PA on Gradescope:

1. Go to gradescope and find Project1 checkpoint/final submission.
    a. If you are submitting with a partner, make sure that the student who submits adds the other student to the assignment.
    b. **Zip Upload:** In the root directory of your Project1, run `./create_submission_zip.sh`. The script will create a zip file named `submission.zip`, then simply drag the zip file to gradescope to finish the submission.
2. **Check the feedback from the autograder and make sure no compile errors appeared.**

Checkpoint Submission (Due Tuesday, May. 19th, 11:59 pm):

When you have completed all of the requirements for the checkpoint submission (Part 1: Pseudo Compression). You should go to gradescope and find Project1 checkpoint submission. **Only the files in your src and test folders will be graded.**

Make sure your code can compile (running 'cmake ..' and 'make' in your project's build directory gives no error)

Final Submission (Due Tuesday, May. 26th, 11:59 pm) :

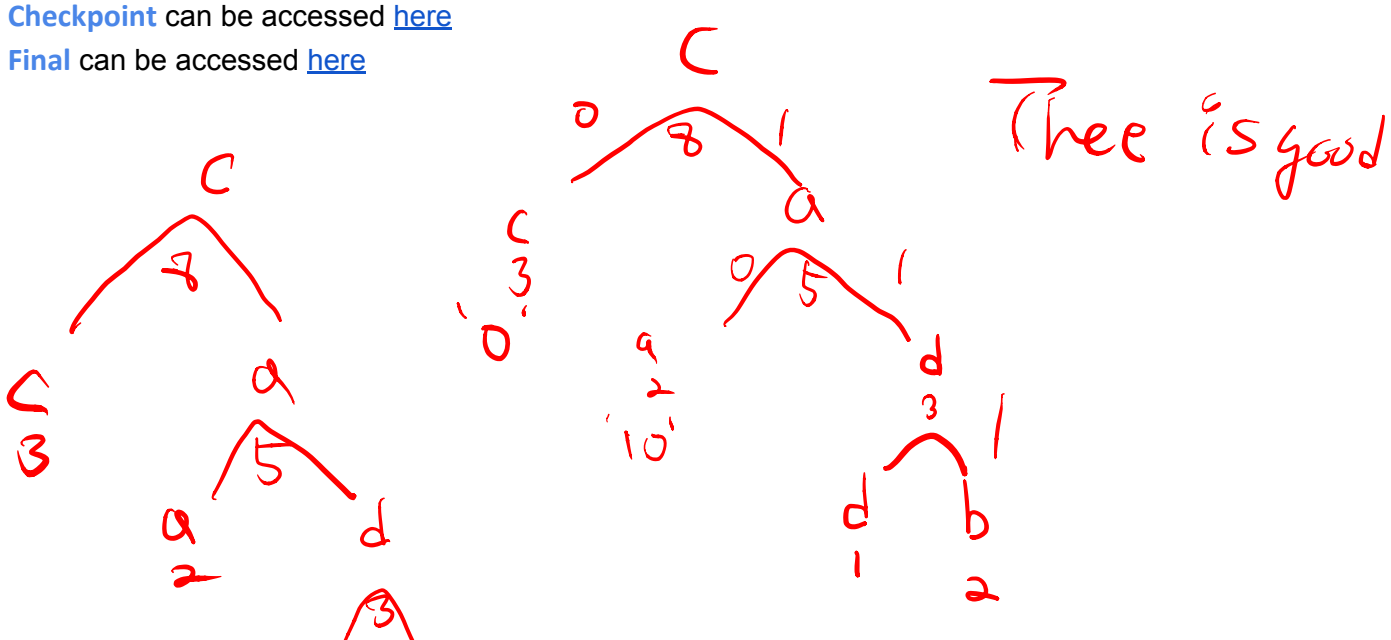
When you have completed all of the requirements for the final submission (all the parts). You should go to gradescope and find Project1 final submission. **Only the files in your src and test folders will be graded.**

encode!

Checkpoint and Final Write-ups

Checkpoint can be accessed here

Final can be accessed here



CSE 100 Project 1 Checkpoint: Pseudo Compression

Checkpoint Deadline: **Tuesday, May 19th, 11:59 pm**

In the first part of this project, you will use a Huffman Coding Tree to implement a pseudo compression without using bitwise i/o (your compressed file will contain chars '0's and '1's to represent the encoding bits).

Your major tasks in this part includes:

1. Implement a Huffman Coding Tree to use as the compression algorithm.
2. Implement a pseudo compress program that "compresses" the given file to an encoded file.
3. Implement a pseudo decompress program that "decompresses" the file generated by the compress program to get the original file back.

1. Write Tests First

You can run the tests by cd-ing into your build directory and running "make && make test". The tests are just normal executables, you can also run them using ./build/path/to/your/test or by using the VSCode debugger. Note that the bit stream implementation is only for the final submission, so the tests for BitInputStream and BitOutputStream are commented out for now.

We only provided simple tests to help you get started writing the tests. Read the function descriptions in part 2 below and write unit tests first before implementing any functions. Then based on your understanding of each function, you should first type out all the function headers in all files to reinforce the understanding of the requirements before you attempt to implement them. Refer to the Testing & Code Coverage part for testing suggestions (in the main document here).

2. Implementing HCTree

Now implement the following methods from `HCNode.hpp`, `HCTree.hpp` and `HCTree.cpp`

Please first read and understand the purpose of each member variable at the top of the file. **Do not modify any public method signature.** You may modify the private helper methods or add any additional member variables or methods if needed. Also, good C++ coding practice requires that you use an hpp file as an interface and put all the implementations in the corresponding cpp file.

We strongly encourage you to unit test each part of your HCTree thoroughly before proceeding to the next part of your HCTree: you will save a significant amount of time in the end if you can prevent having to debug your HCTree as a whole. Refer to the Testing part for testing suggestions.

For this part of the assignment you need to implement the following methods in the following files (descriptions of methods are in the the files):

HCNode.hpp

- `bool operator()(HCNode*& lhs, HCNode*& rhs) const`

HCTree.hpp, HCTree.cpp

- `HCTree()`
- `void build(const vector<unsigned int>& freqs)`
- `void encode(byte symbol, ostream& out) const`
- `byte decode(istream& in) const`
- `~HCTree()`

Note: The following 2 functions in HCTree are not required for checkpoint submission. You will implement them in the final submission:

- `void encode(byte symbol, BitOutputStream& out) const`
- `byte decode(BitInputStream& in) const`

```
gdb --args build/src/compress --ascii data/bitstream_1.txt bitstream_1_compressed.txt
```

3. Implementing Compression and Decompression Program

For this part of the assignment you need to implement the following methods in the following files (descriptions of methods are in the the files):

compress.cpp

- `void pseudoCompression(const string& inFileName, const string& outFileName)`

uncompress.cpp

- `void pseudoDecompression(const string& inFileName, const string& outFileName)`

First, compile the empty compress.cpp and uncompress.cpp, and check the following:

1. This is the path to your compress executable: **./build/src/compress**
2. This is the path to your uncompress executable: **./build/src/uncompress**

When implementing the compress and uncompress program, make sure your program follows good object-oriented design: all the compression/decompression parts should be done by just calling functions of HCTree.

Command line options parsing

The following commands with option "--ascii" should make your program perform pseudo compression and decompression using ascii symbols '0's and '1's:

```
./build/src/compress --ascii data/toCompress.txt compressed.txt

./build/src/uncompress --ascii data/compressed.txt decompressed.txt
```

Whereas running without the ascii option will do the true compression using the bit stream that you will implement in the final submission.

Compression

Now use your Huffman Coding Tree to implement a program that performs pseudo compression on a given file. In checkpoint submission, the compress program should take three arguments: the “ascii” option, the name of the file to be compressed, and the name of the compressed file:

```
./build/src/compress --ascii data/toCompress.txt compressed.txt
```

First, your program should read the first file and build the Huffman Coding Tree based on the frequency of each char in that file. Then your program should open the compressed file (create it if it does not exist) and first write a header that can help the uncompress program to reconstruct the same tree. Your header in the checkpoint should have 256 lines, with each line being the frequency of the ASCII symbol with the value of the line number. Then, write the encoding bits (represented as chars of either '0's or '1's) of all the chars in the original file to the compressed file.

You should implement this file from scratch so make sure you understand how to properly do i/o in C++. Online resources can be helpful. Arguments parsing and file checking has already been handled for you in main. And all your implementation for checkpoint in compress.cpp should be done in `pseudoCompress()` and any custom helper functions you add.

Once you finished implementing the compress program, you should compare the output with with the reference solution:

```
./solution-compress.executable --ascii data/toCompress.txt compressed.txt
```

Given the same original file, it is expected that your compressed file is always the same as the one from the reference solution, since you should be using the same tie breaking rules when building the HCTree.

Note that because we are using ascii chars to represent bits, the compressed file is actually larger than the original, which is why we called it a *pseudo* compress program. Later in part 2, with the help of bitwise i/o, you will start implementing the true compression program.

Decompression null character

In checkpoint submission, the decompress program should take three arguments: the "ascii" option, the name of the compressed file, and the name of the decompressed file:

```
./build/src/uncompress --ascii compressed.txt decompressed.txt
```

First, your program should read the first file and build the Huffman Coding Tree based on the header of the file. Then your program should open the decompressed file (create it if it does not exist) and write decoded chars of the compressed file to the output file, such that the output file contains the exact same content as the original file.

Arguments parsing and file checking is already handled for you in main. And all your implementation for checkpoint in uncompress.cpp should be done in `pseudoDecompress()` and any custom helper functions you add.

You should run the diff command to see if your output file is the same as the original one:

```
diff data/toCompress.txt decompressed.txt
```

Please refer to the testing cases part for guidance on making your own testing files.

Testing Cases and Grading Breakdown

We've provided some test files that we are using in the autograder for you, make sure you read "Diffing output" to understand the requirements. You can find the provided test files in the data folder.

Note that your compression program should work regardless of the size and type of the files. You can only assume the input file is less than 4GB, which means a particular symbol may appear 4 billion times.

: provided test files in data folder (file with name file_1.txt etc.)
: makeup credit (if you received less than the maximum #points, these points will apply)

Checkpoint (30 pts)

In checkpoint, to get full credits for each testing file ("File 1" etc), the compressed and uncompressed files from your program should match the ones from the reference solution.

Testing Cases	Description	Points
File 1	An empty file with no content	2
File 2	A file contains only one repeating single kind of chars (all newlines)	3
File 3	A file with alphabet letters	5
File 4	A file with symbols possibly from extended ASCII table, with ASCII value larger than 127.	5
File 5	The novel *War and Peace*	5
File 6	Non-text files such as binary files (executable), image, and video	5
Valgrind	No memory leaks in compress.cpp and uncompress.cpp	3
Code formatting	Passing the autoformatting check	1
Student tests	Passing your own tests	1
Code coverage	90% of line coverage for encoder	10

```
#ifndef HCNODE_HPP
#define HCNODE_HPP
#include <iostream>
typedef unsigned char

using namespace

/** A class, instances of which are nodes in an
    HCTree.
 */
class
  public:
    unsigned int          // the frequency of the
                             symbol
                          // byte in the file we're keeping
                             track of

                          // pointer to '0' child

                          // pointer to '1' child
                          // pointer to parent

  /* Constructor that initialize a HCNode */
        unsigned int
              *           *
            *
```

```
/* For printing an HCNode to an ostream. Possibly
 useful for debuging */
        & operator          & const        &

        & operator          &     const         &
              "["                  ","
            int            "]"
      return


/* Comparator of HCNode pointer. In priority queue
 , HCNode ptr with lower count has higher priority,
 and if count is the same, HCNode ptr with larger
 symbol has higher priority.
*/
struct
      /* TODO */
      bool operator          *&          *&
             const    return false

#endif      // HCNODE_HPP
```

CSE 100 Discussion 7

Project 1 Checkpoint: Huffman Encoding and Decoding

Overview/Table of Contents

Recommendations

- This will take considerably longer than the PAs. Make sure to start early!
- Copying and pasting is evil!
- Use helper methods!
- GDB GDB GDB!!!!!! (backtrace!)

NOTES

No hidden tests in checkpoint, some in final

Running gdb:

```
gdb --args build/src/compress --ascii data/bitstream_1.txt bitstream_1_compressed.txt
```

Resources

- Stepik
- Videos
- Visualization (does not follow exact same format as our assignment)
- Piazza FAQs (and other posts!)

Project 1 Checkpoint

Part 1: Write Tests First

Part 2: Implementing HCTree

Part 3: Implementing Compression and Decompression

File Structure

Testing: Write Tests First!

How do I test compress.cpp and uncompress.cpp?

- You cannot write unit tests for files with a main method
- To reduce complexity write helper methods/files that are testable!
- line coverage testing with only check coverage of encoder directory for checkpoint (encoder + bitstream dirs for final)

Shell Scripting is Your Friend!

Creating a script to run multiple files and compare (diff)

```
diff data/toCompress.txt decompressed.txt
```

Is there a way we could automate diffing multiple runs of our program at once?

Yes! You can create a script to test multiple files and output whether or not there was a difference between the solution and your program.

Using Huffman’s Algorithm to Construct a Tree

Overview: Methods to Implement

`HCNode.hpp`

- `bool operator()(HCNode*& lhs, HCNode*& rhs) const`

`HCTree.hpp, HCTree.cpp`

- `HCTree()`
- `void build(const vector<unsigned int>& freqs)`
- `void HCTree::encode(byte symbol, ostream& out) const`
- `byte HCTree::decode(istream& in) const { return ' '; }`
- `~HCTree()`

What is an HCNode?

`unsigned int count` → the frequency of the symbol

`byte symbol` → byte in the file we're keeping track of

`HCNode* c0` → pointer to '0' child

`HCNode* c1` → pointer to '1' child

`HCNode* p` → pointer to parent

HCNode Comparator

```
bool operator()(HCNode*& lhs, HCNode*& rhs) const { return false; }
```

Comparing `HCNode*& lhs` and `HCNode*& rhs`

- Lower `count` has higher priority
- If `count` the same, `HCNode` with larger `symbol` has higher priority

Remember to add the comparator! See example here

```
template <class T, class Container = vector<T>,

class Compare = less<typename Container::value_type> > class priority_queue;
```

What is an HCTree?

`HCNode* root` → the root of HCTree

`vector<HCNode*> leaves` → a vector storing pointers to all **leaf** HCNodes

HCTree Constructor: HCTree()

Description: Initializes a new empty HCTree.

What do we need to do to initialize an empty HCTree?

HCTree Constructor: HCTree()

Description: Initializes a new empty HCTree.

What do we need to do to initialize an empty HCTree?

- need to initialize `root` and `leaves`
- `leaves` is a `vector` of `HCNode*`

HCTree: build

```
void build(const vector<unsigned int>& freqs)
```

Build the HCTree from the given frequency vector. You can assume the vector must have size **256** and each value at index i represents the frequency of char with ASCII value i. Only non-zero frequency symbols should be used to build the tree. The leaves vector must be updated so that it can be used in encode() to improve performance.

When building the HCTree, you should use the following tie-breaking rules to match the output from reference solution in checkpoint:

1. HCNode with lower count should have higher priority. If count is the same, then HCNode with a larger ascii value symbol should have higher priority. (This should be already defined properly in the comparator in HCNode.hpp)
2. When popping two highest priority nodes from PQ, **the higher priority node will be the 'c0' child of the new parent HCNode**.
3. The symbol of any parent node should be taken from its 'c0' child.

HCTree: build

To understand this method, let’s look at the pseudo code and walk through an example!

Huffman’s Algorithm

Huffman's Algorithm: Bottom-up Construction

0. Determine the count of each symbol in the input message.
1. Create a forest of single-node trees containing symbols and counts for each non-zero-count symbol.
2. Loop while there is more than 1 tree in the forest:
    2a. Remove the two lowest count trees
    2b. Combine these two trees into a new tree (summing their counts).
    2c. Insert this new tree in the forest, and go to 2.
3. Return the one tree in the forest as the Huffman code tree.

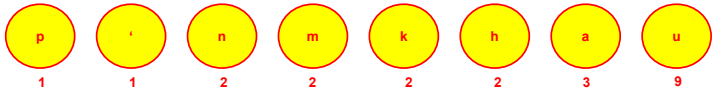
0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
    2a. Remove 2 lowest count trees
    2b. Combine trees
    2c. Insert new tree in forest, go to 2
3. Return final tree.

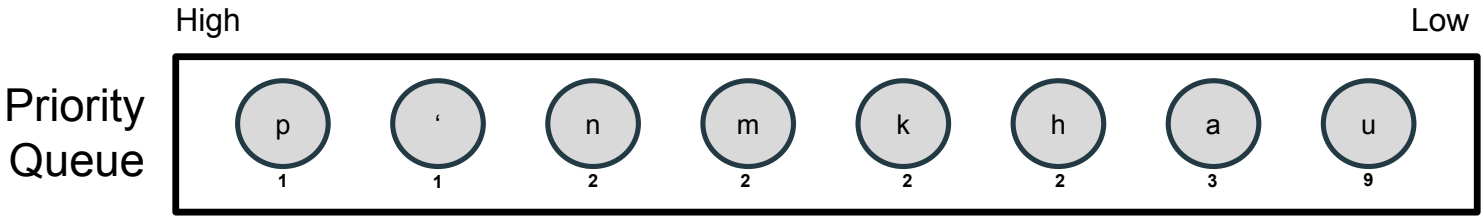
Input Message: humuhumunukunukuapua’a

h	u	m	n	k	a	p	‘
2	9	2	2	2	3	1	1

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
    2a. Remove 2 lowest count trees
    2b. Combine trees
    2c. Insert new tree in forest, go to 2
3. Return final tree.

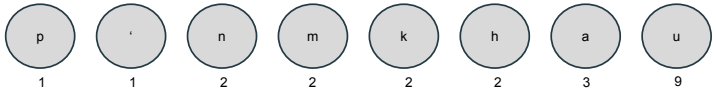
Forest of Trees

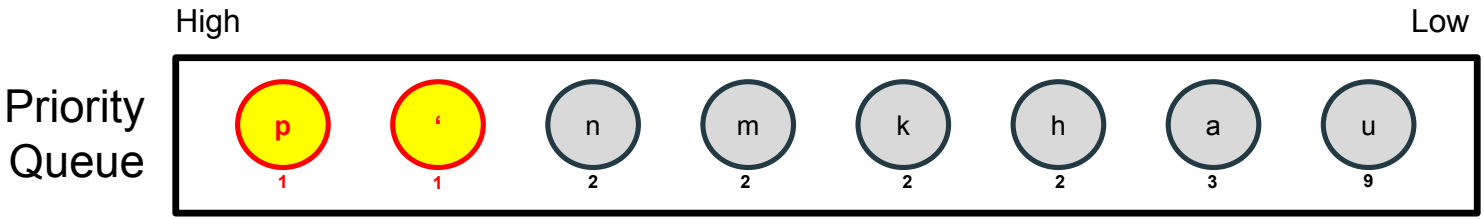




0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

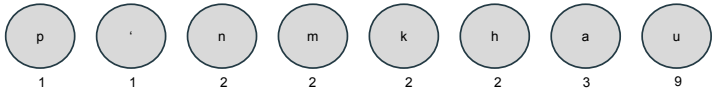
Forest of Trees

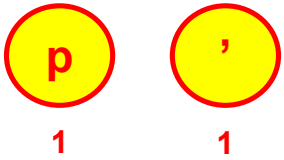


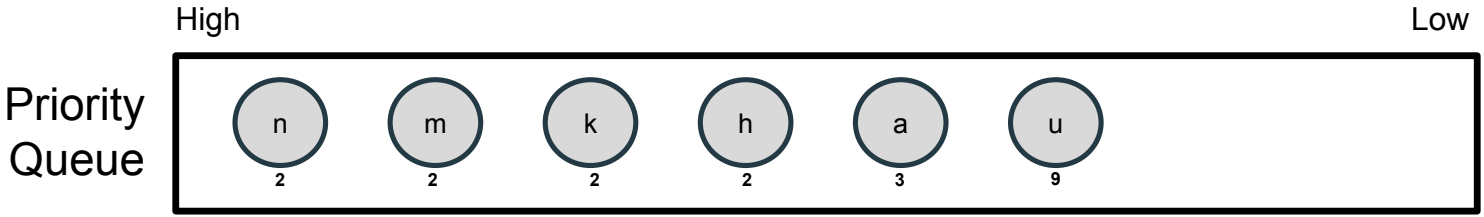


0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

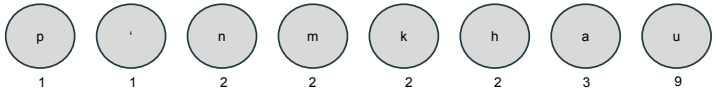


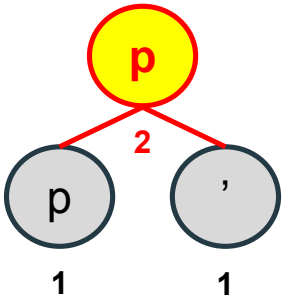


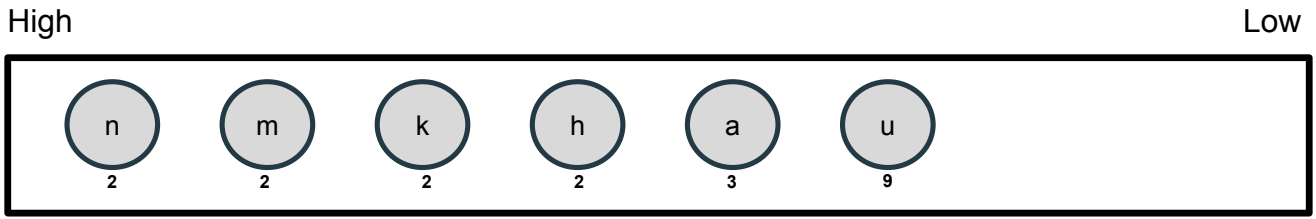


0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

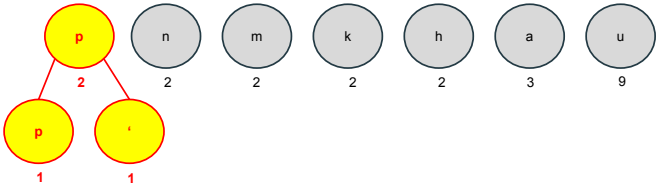






0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

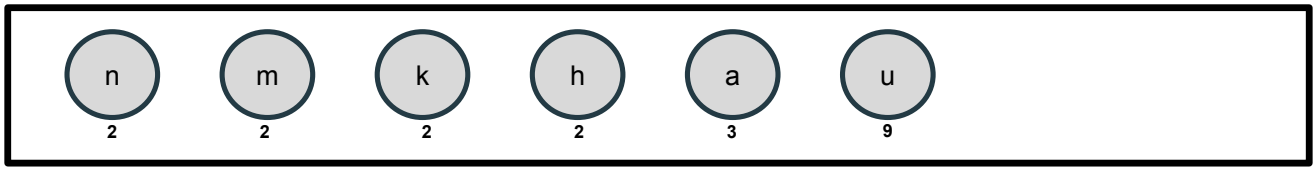
Forest of Trees



High

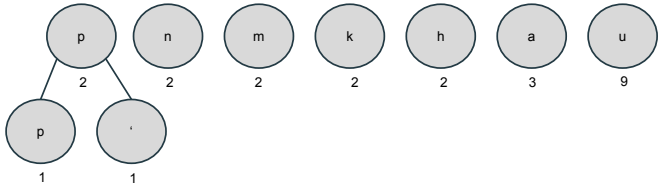
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

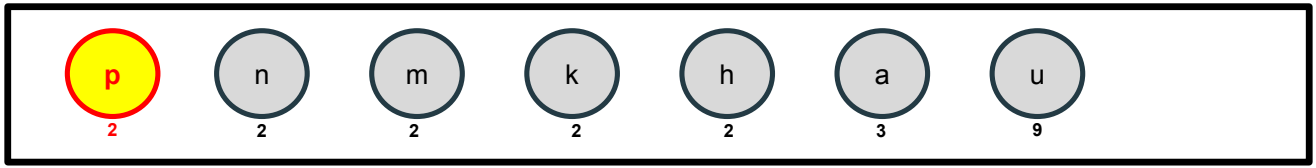
Forest of Trees



High

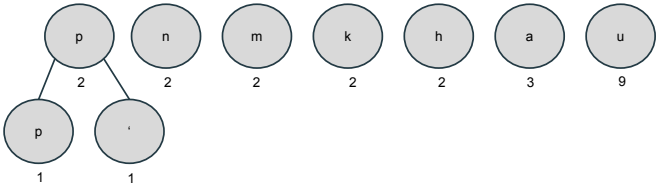
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
    2a. Remove 2 lowest count trees
    2b. Combine trees
    2c. Insert new tree in forest, go to 2
3. Return final tree.

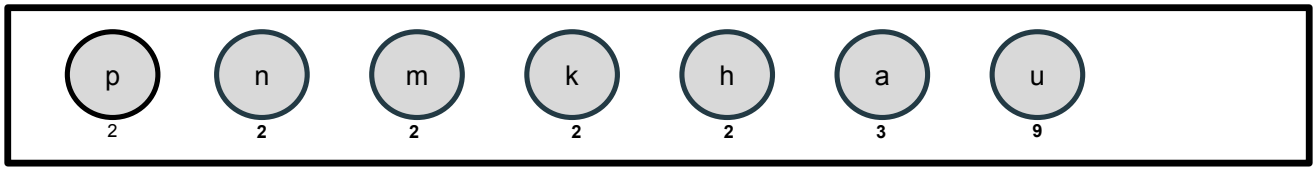
Forest of Trees



High

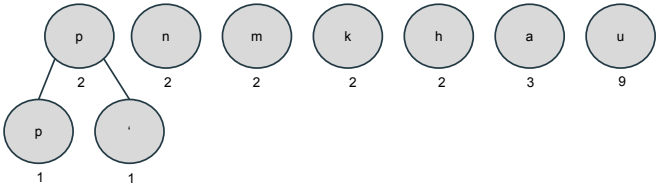
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

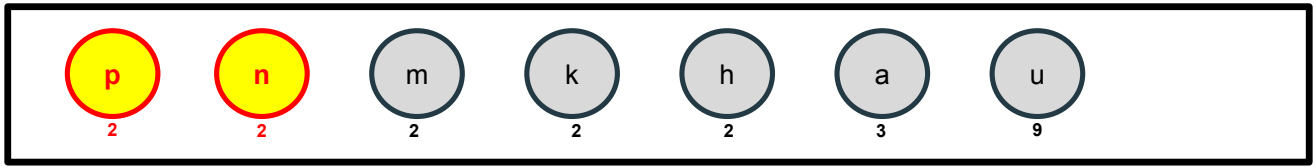
Forest of Trees



High

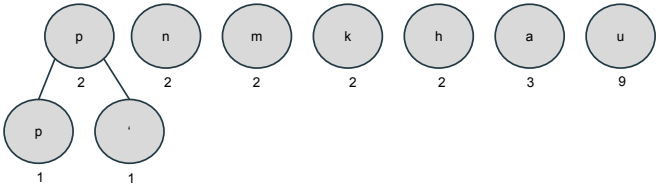
Low

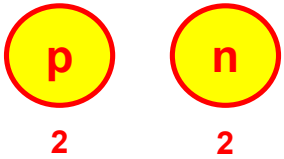
Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

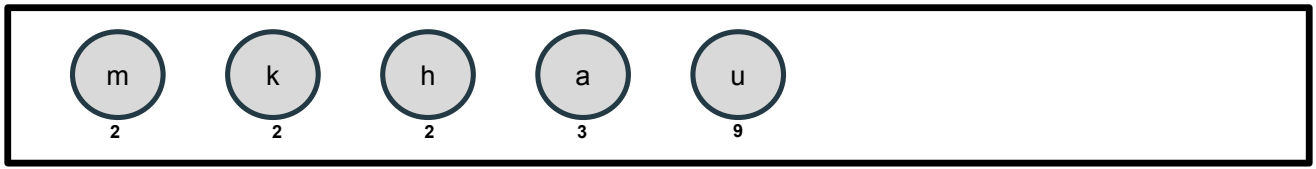




High

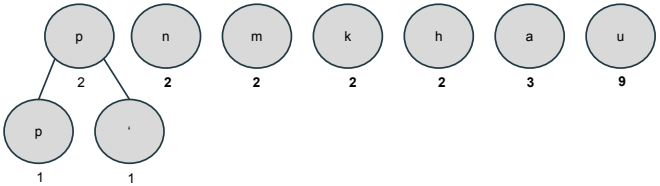
Low

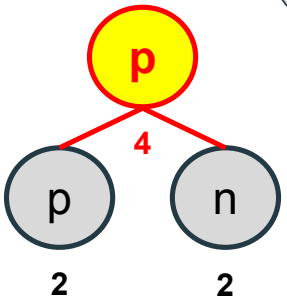
Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

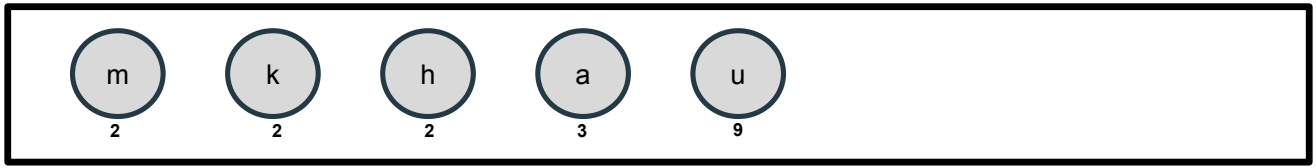




High

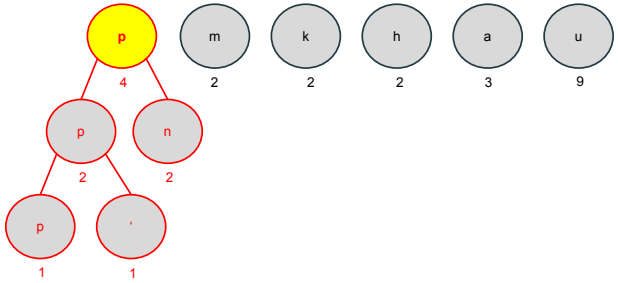
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

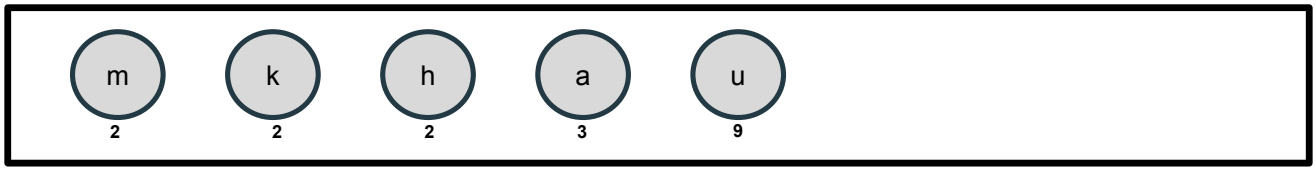
Forest of Trees



High

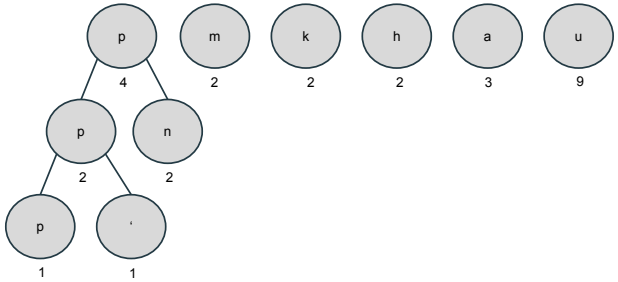
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

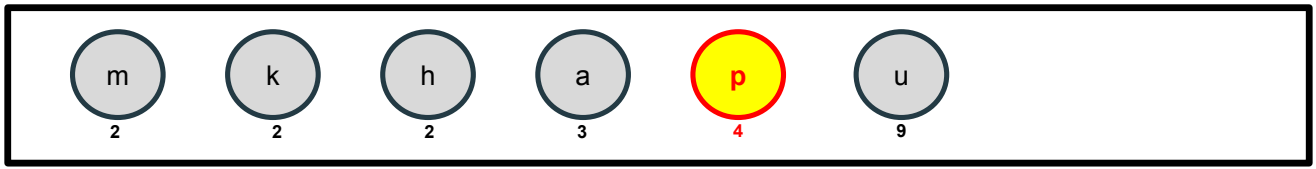
Forest of Trees



High

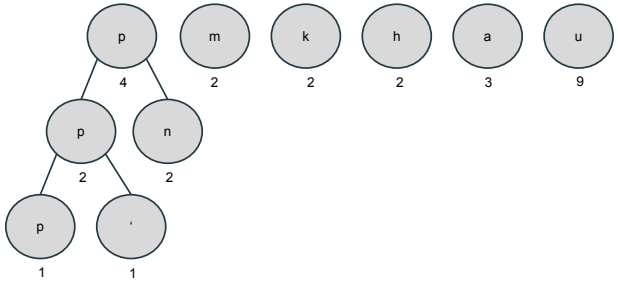
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
    2a. Remove 2 lowest count trees
    2b. Combine trees
    2c. Insert new tree in forest, go to 2
3. Return final tree.

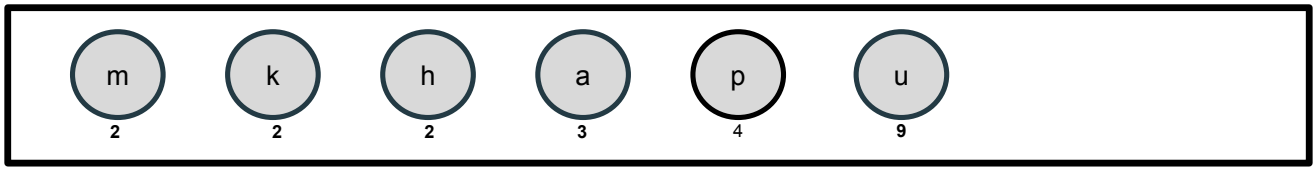
Forest of Trees



High

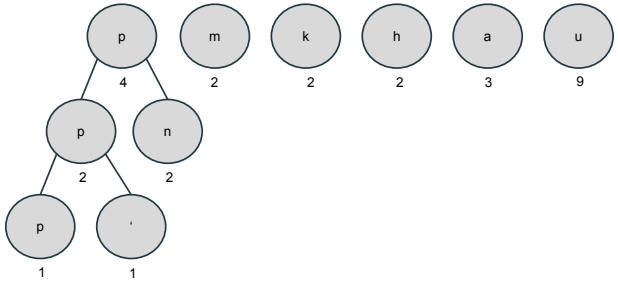
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

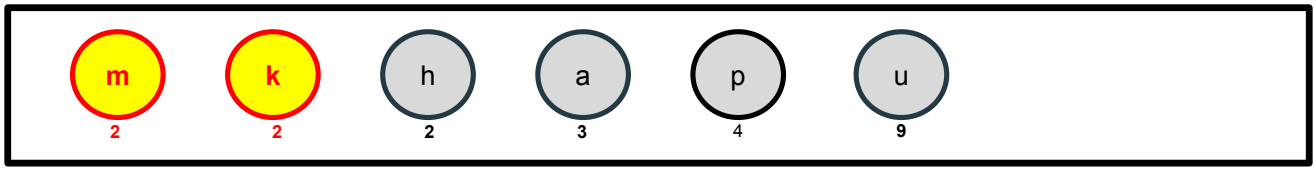
Forest of Trees



High

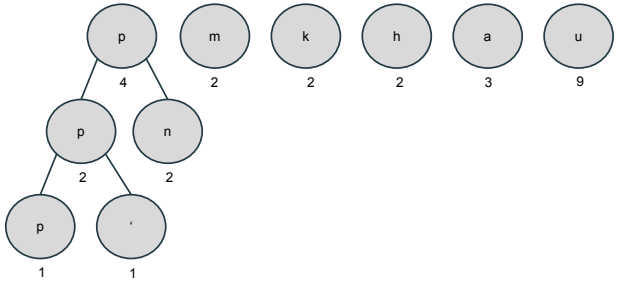
Low

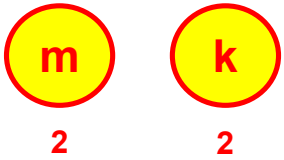
Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

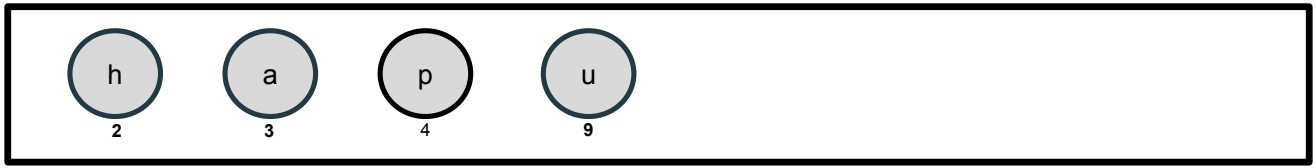




High

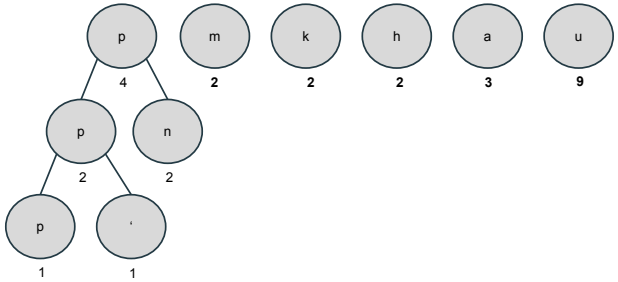
Low

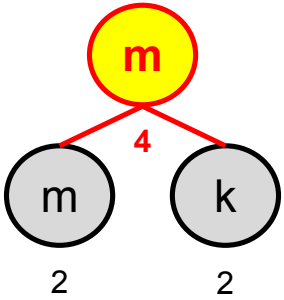
Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees





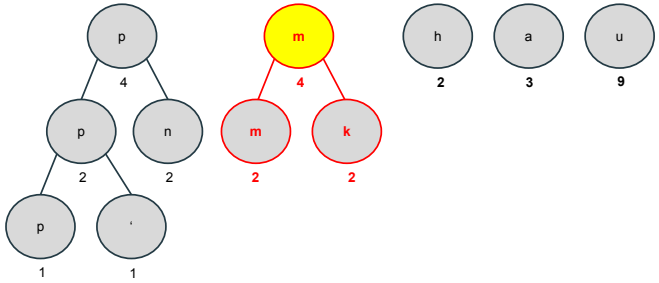
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

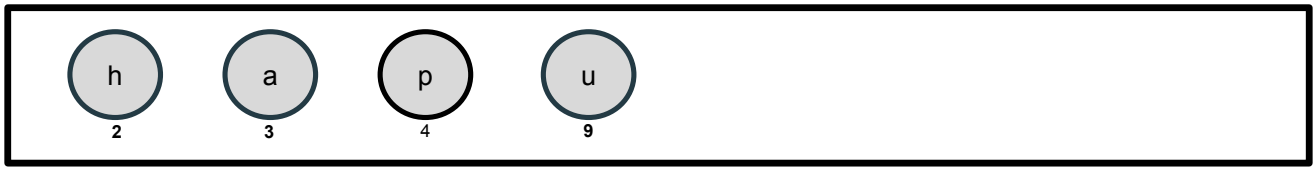
Forest of Trees



High

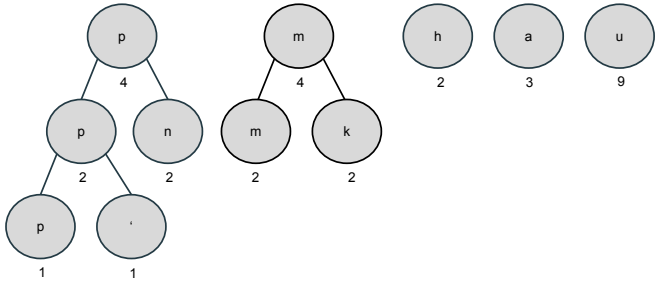
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

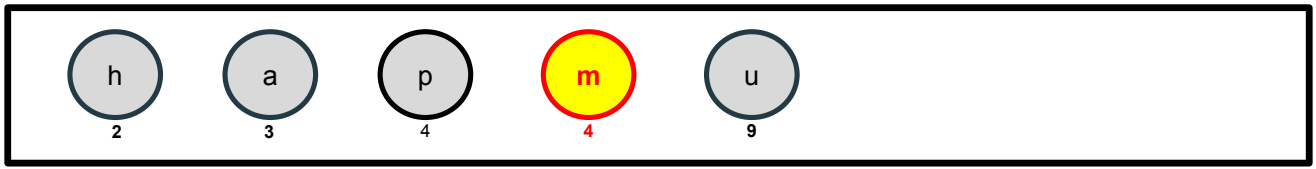
Forest of Trees



High

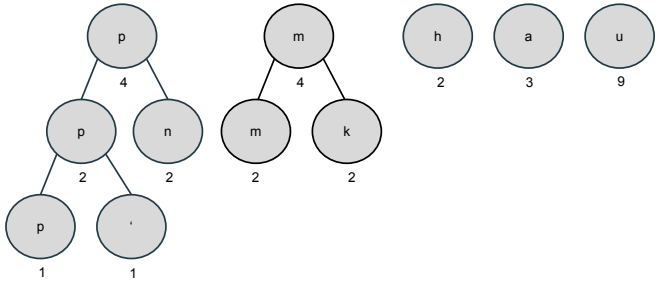
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

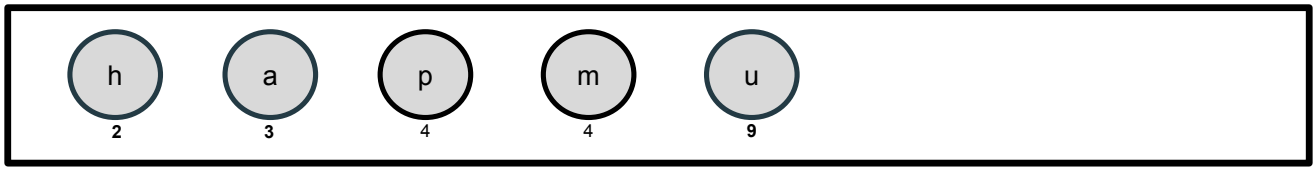
Forest of Trees



High

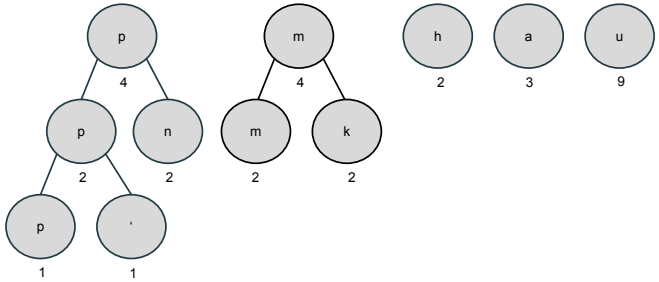
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

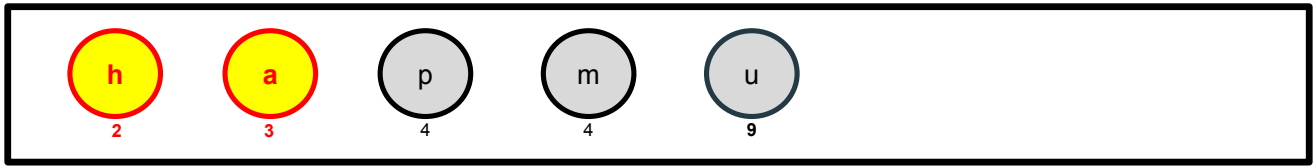
Forest of Trees



High

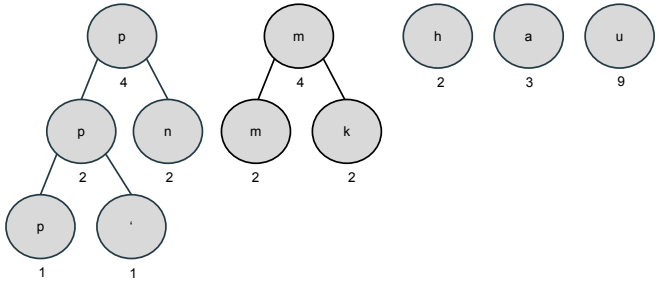
Low

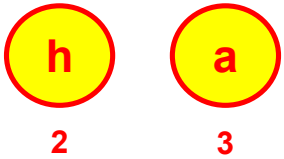
Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees





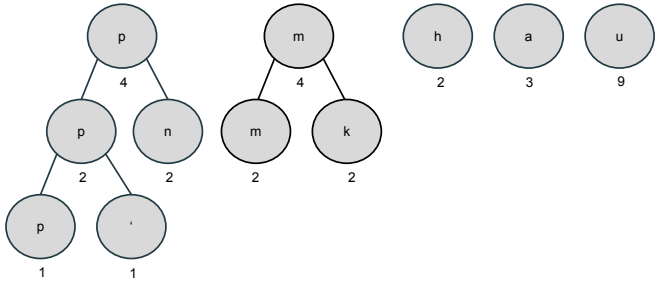
High

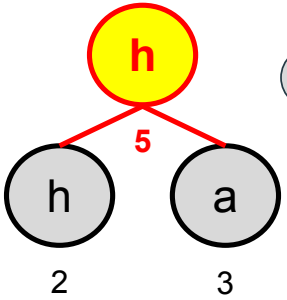
Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees





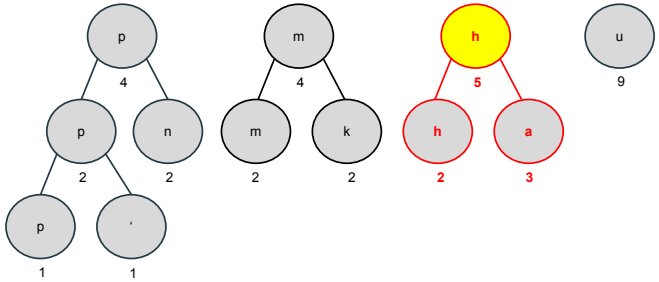
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



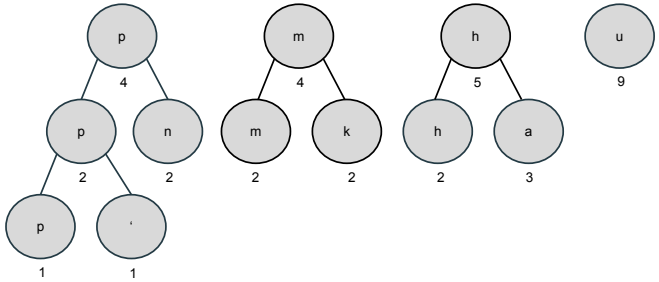
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

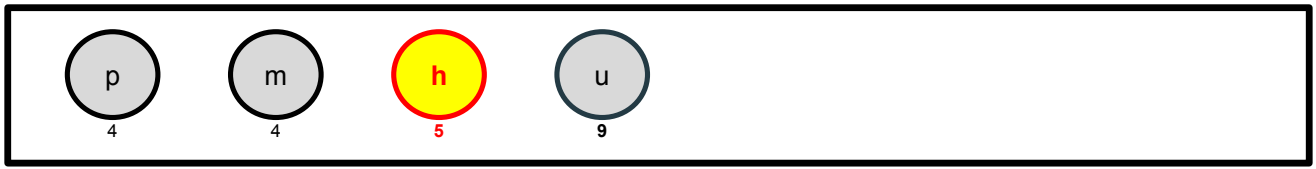
Forest of Trees



High

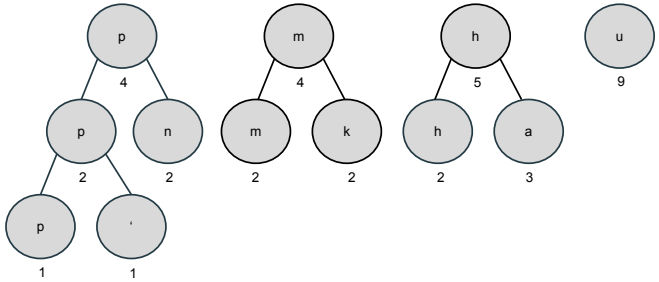
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

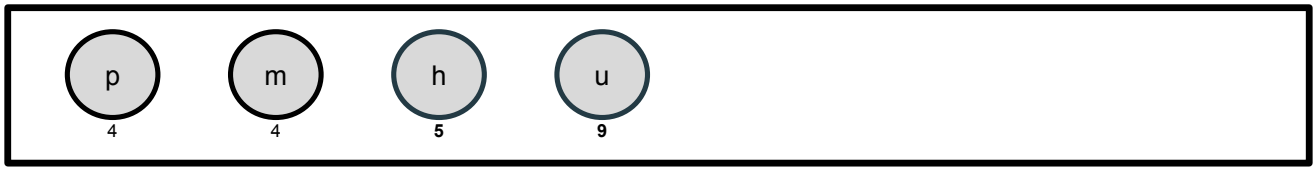
Forest of Trees



High

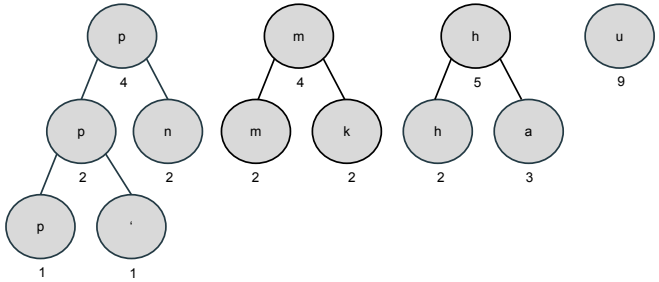
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

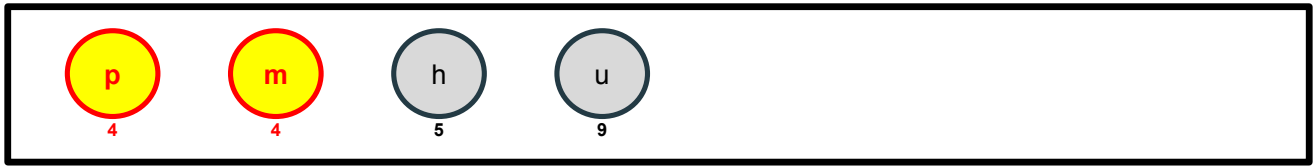
Forest of Trees



High

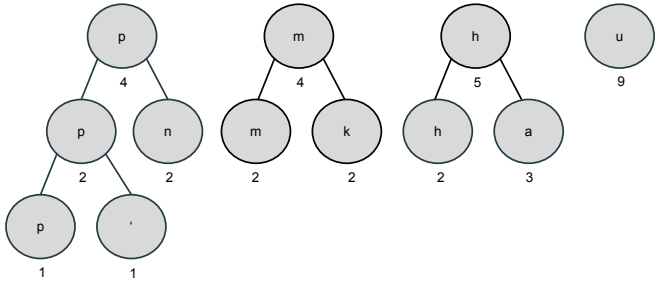
Low

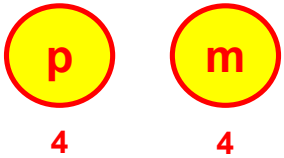
Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees





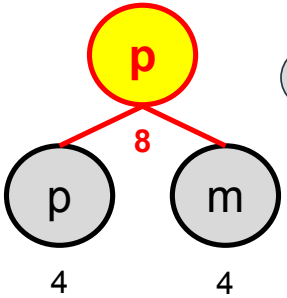
High

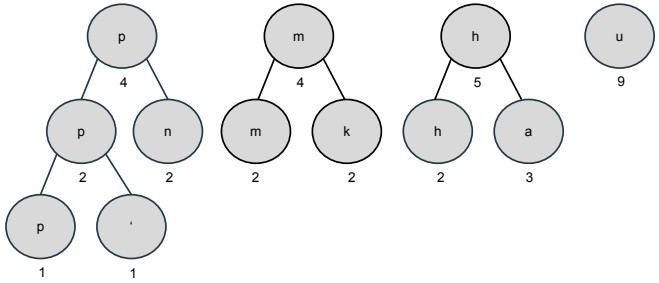
Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees





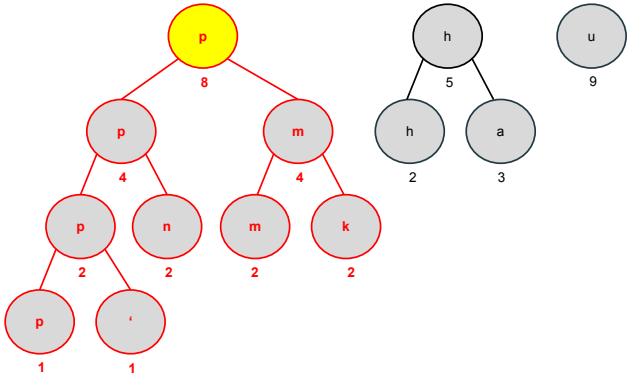
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



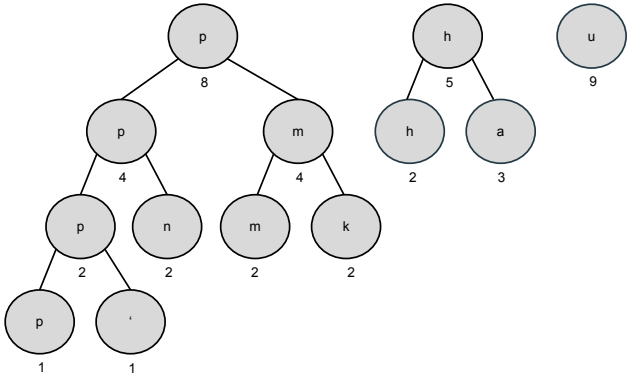
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



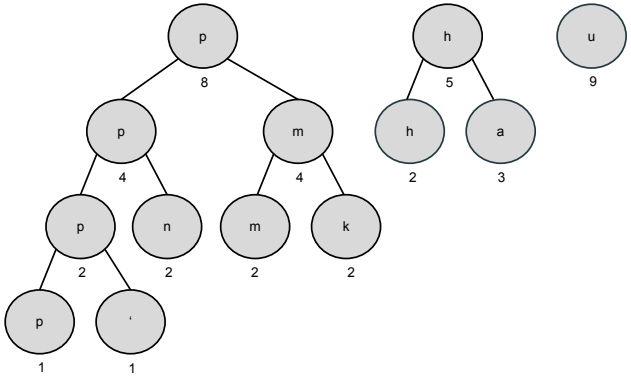
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
    2a. Remove 2 lowest count trees
    2b. Combine trees
    2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



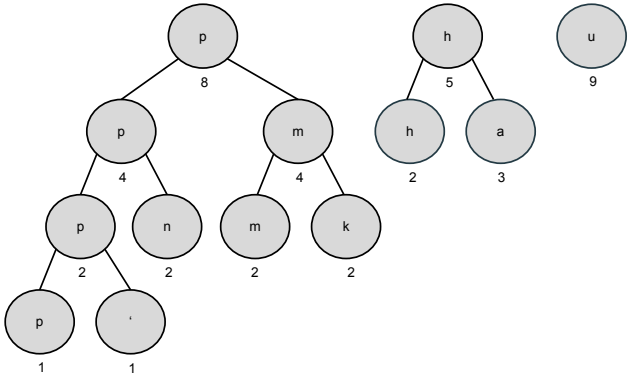
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



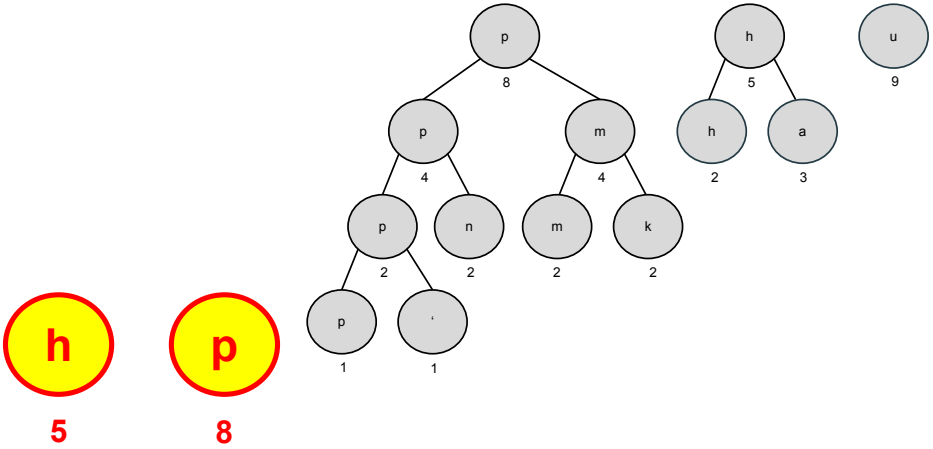
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

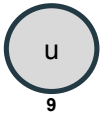


h **p**

5 **8**

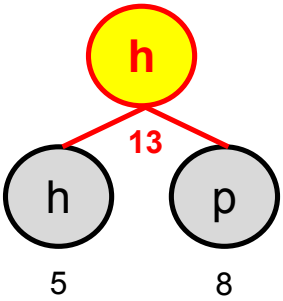
High Low

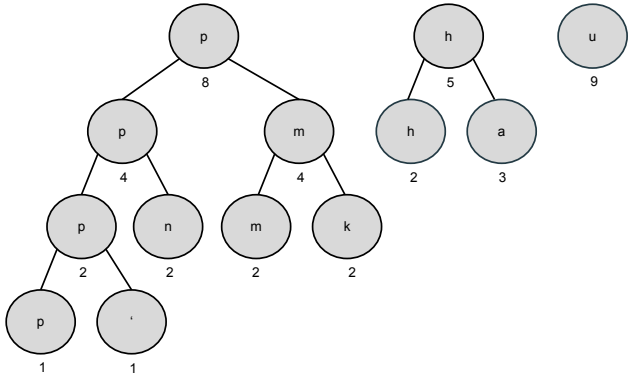
Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

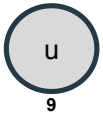




High

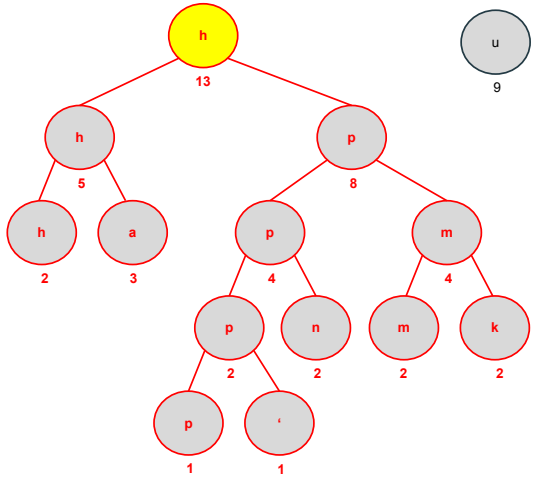
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

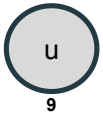
Forest of Trees



High

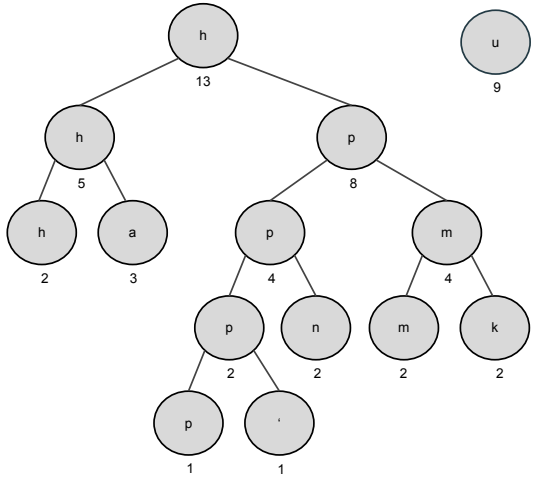
Low

Priority Queue



0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



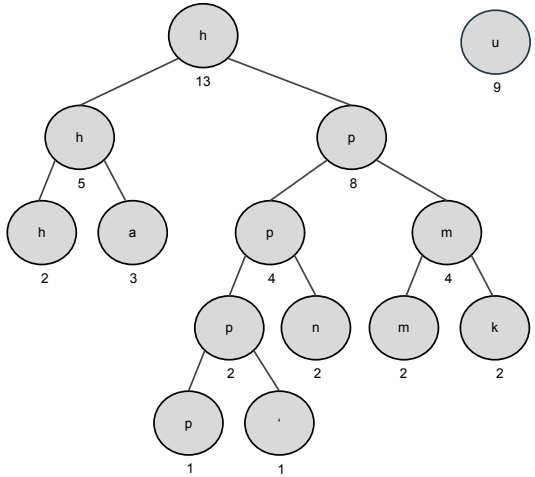
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



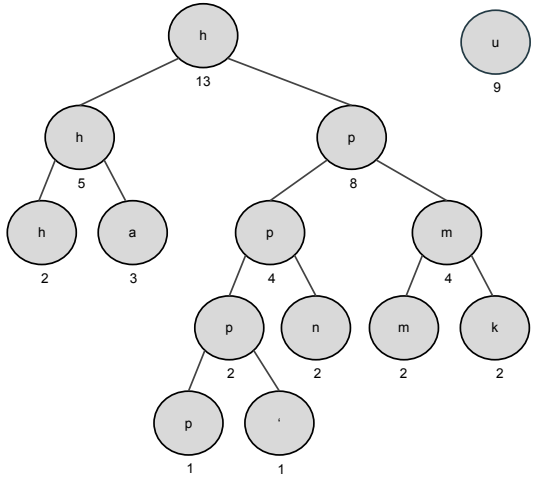
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



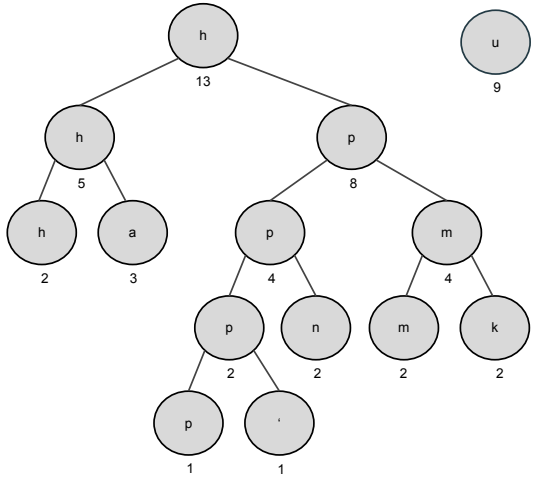
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



u **h**

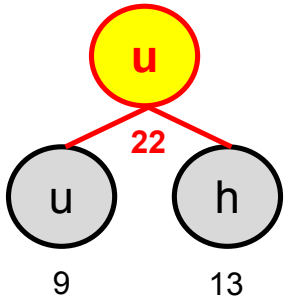
9 **13**

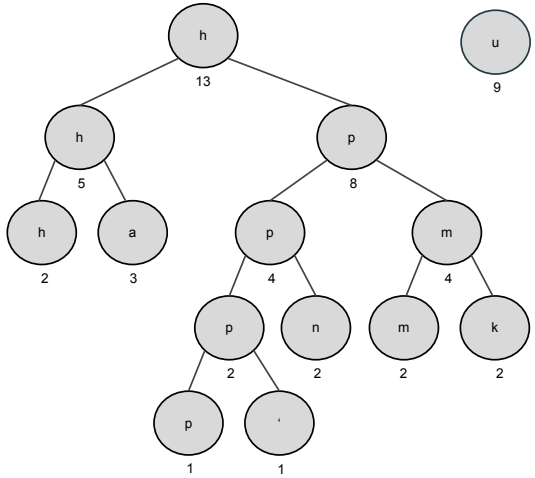
High Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees





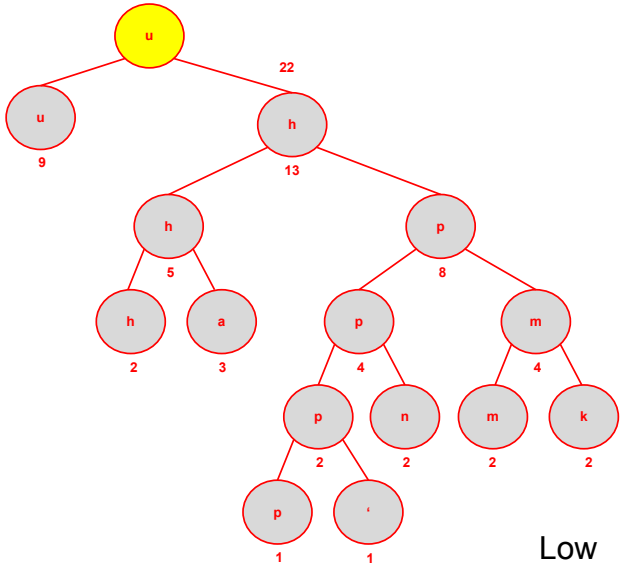
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
   2a. Remove 2 lowest count trees
   2b. Combine trees
   2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



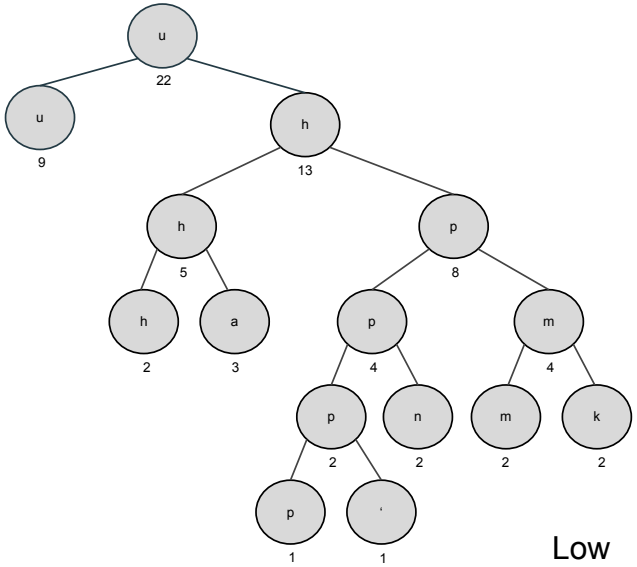
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
    2a. Remove 2 lowest count trees
    2b. Combine trees
    2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees



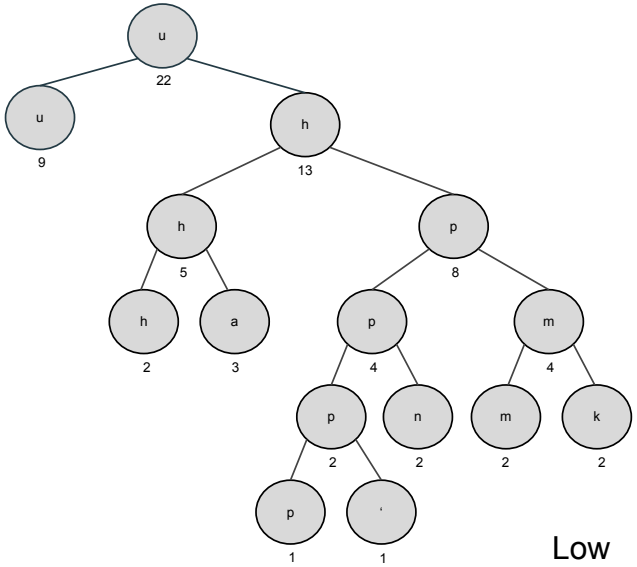
High

Low

Priority Queue

0. Determine count
1. Create forest of single-node trees
2. Loop while > 1 tree in forest:
    2a. Remove 2 lowest count trees
    2b. Combine trees
    2c. Insert new tree in forest, go to 2
3. Return final tree.

Forest of Trees

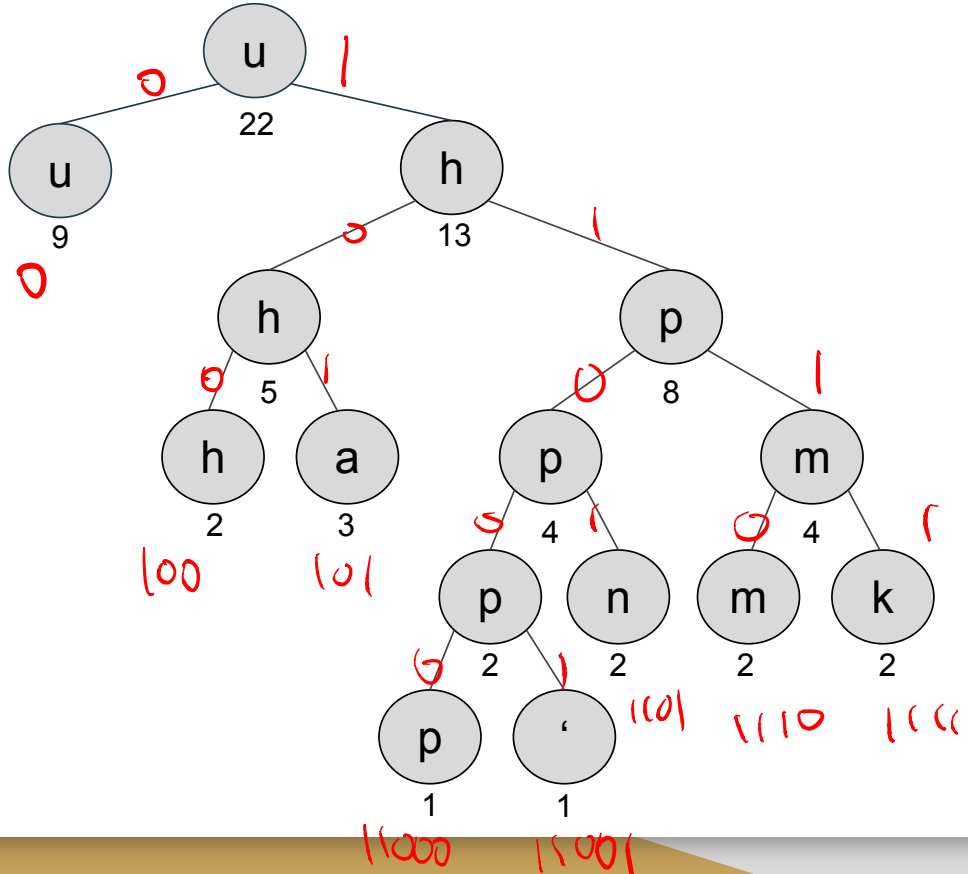


High

Low

Priority Queue

humankap i



Huffman's Algorithm: Bottom-up Construction

Why do we always remove the two lowest count trees?

What do we achieve by doing this?

Huffman's Algorithm: Bottom-up Construction

Why do we always remove the two lowest count trees?
What do we achieve by doing this?

Because the ones you pick first will end up at a lower level in the tree, so their encoding will be longer. So you don't want the first ones that you pick to be super frequent chars.

HCTree Destructor: ~HCTree()

Description: Delete all objects on the heap to avoid memory leaks.

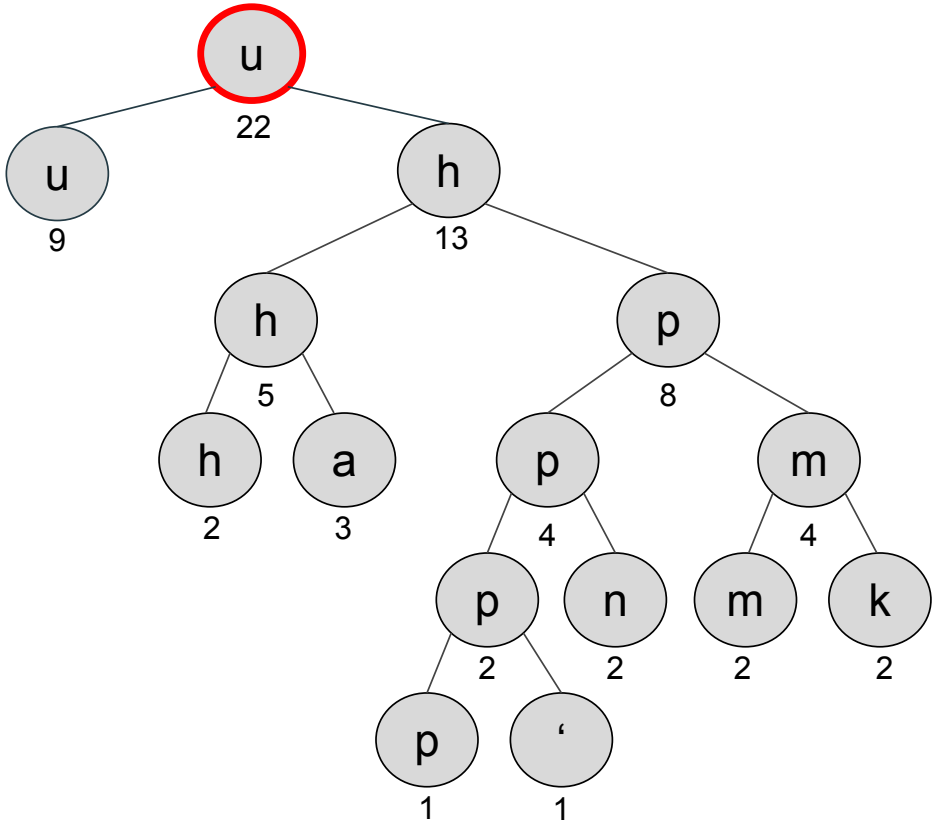
What is the best way to do this?

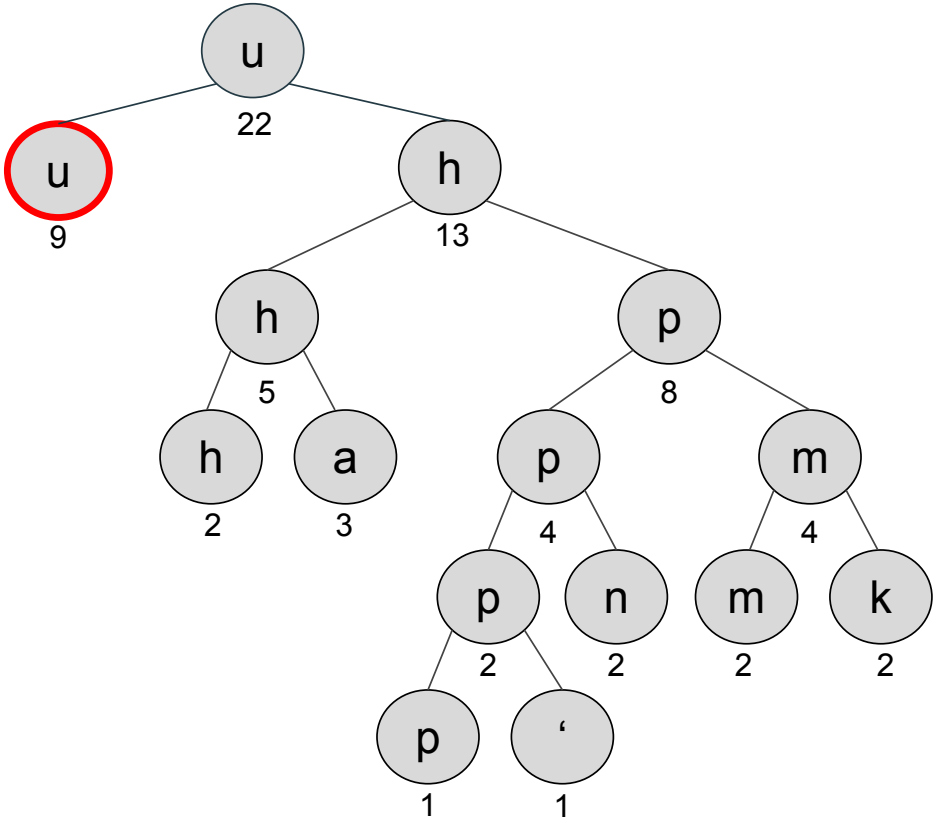
HCTree Destructor: ~HCTree()

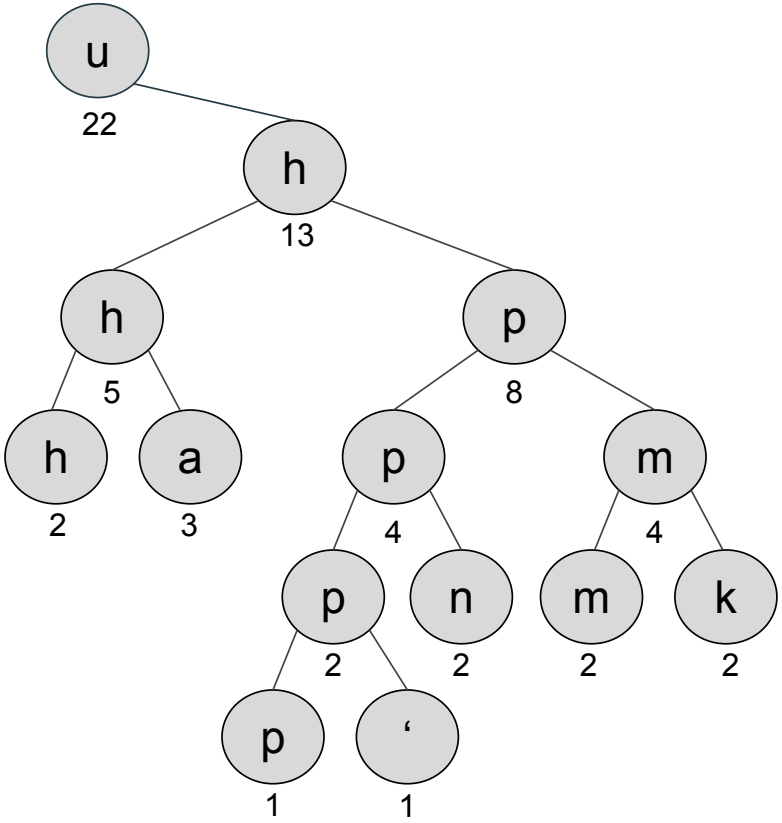
Description: Delete all objects on the heap to avoid memory leaks.

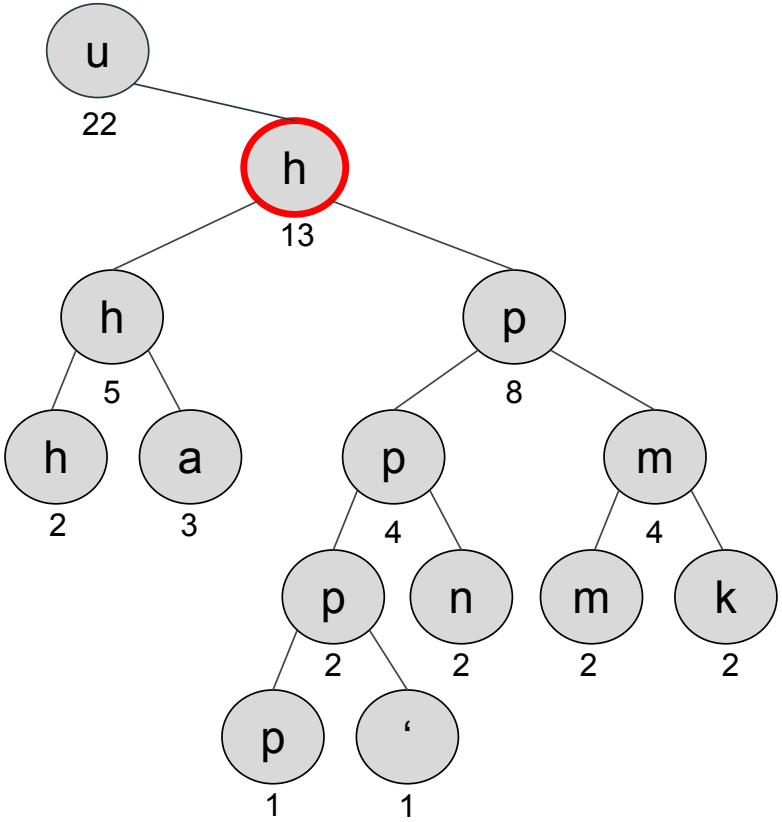
What is the best way to do this?

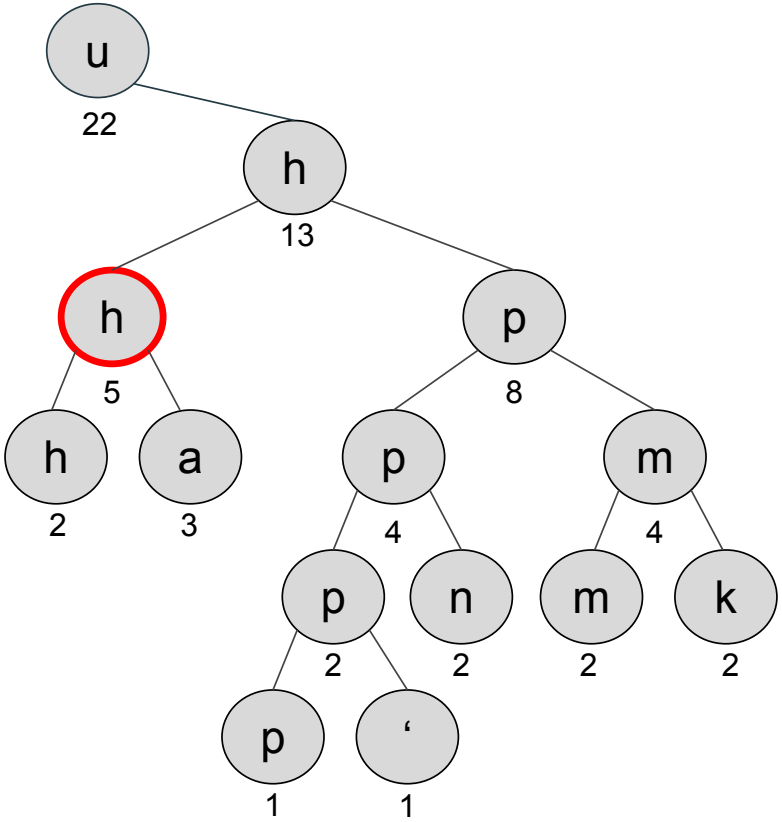
Recursively!

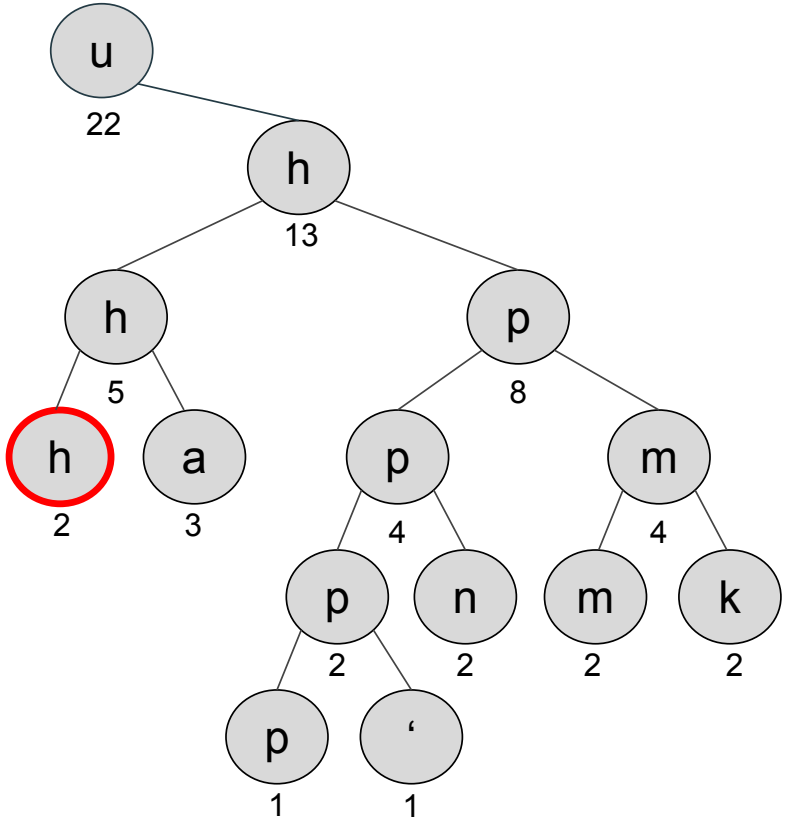

u
22
u
9
h
13
h
5
p
8
h
2
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

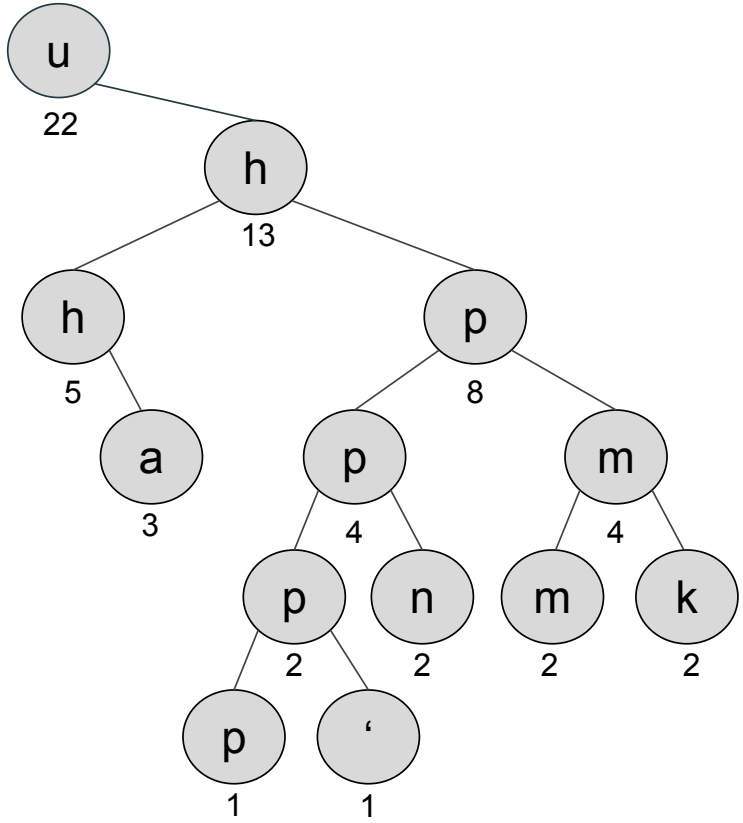

u
22
u
9
h
13
h
5
p
8
h
2
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

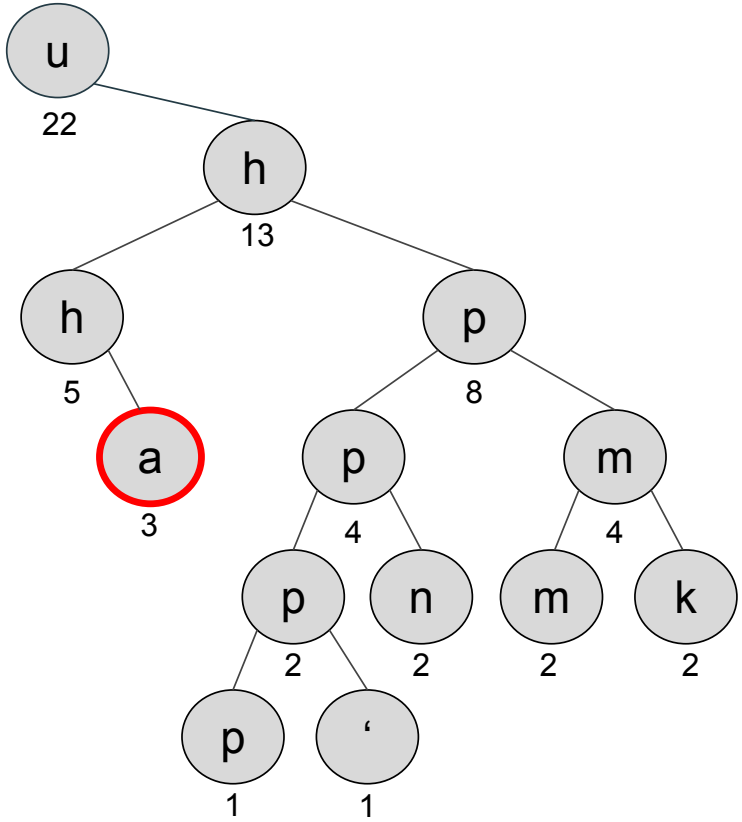

u
22
h
13
h
5
p
8
h
2
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

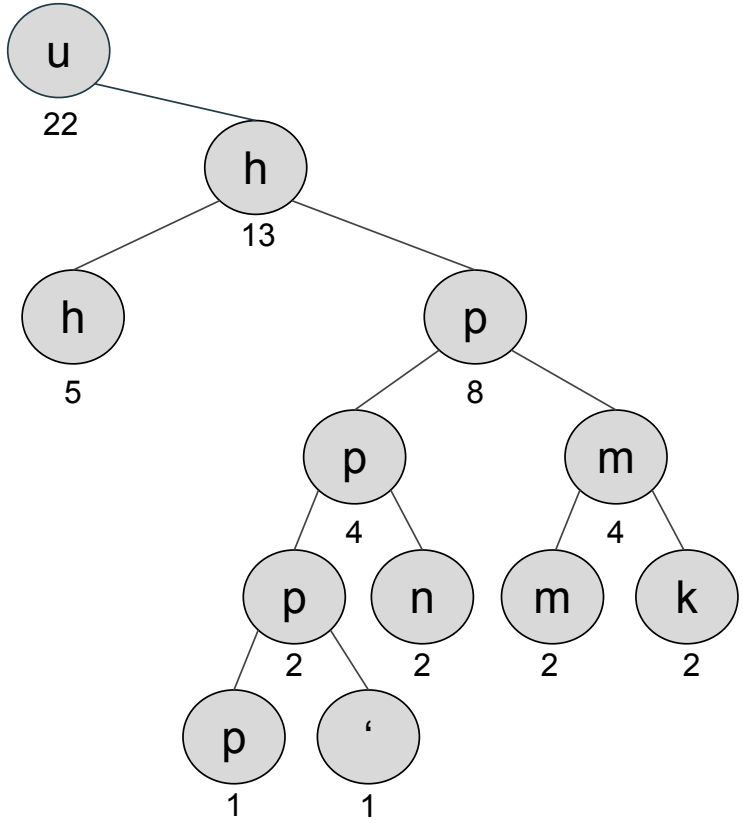

u
22
h
13
h
5
p
8
h
2
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

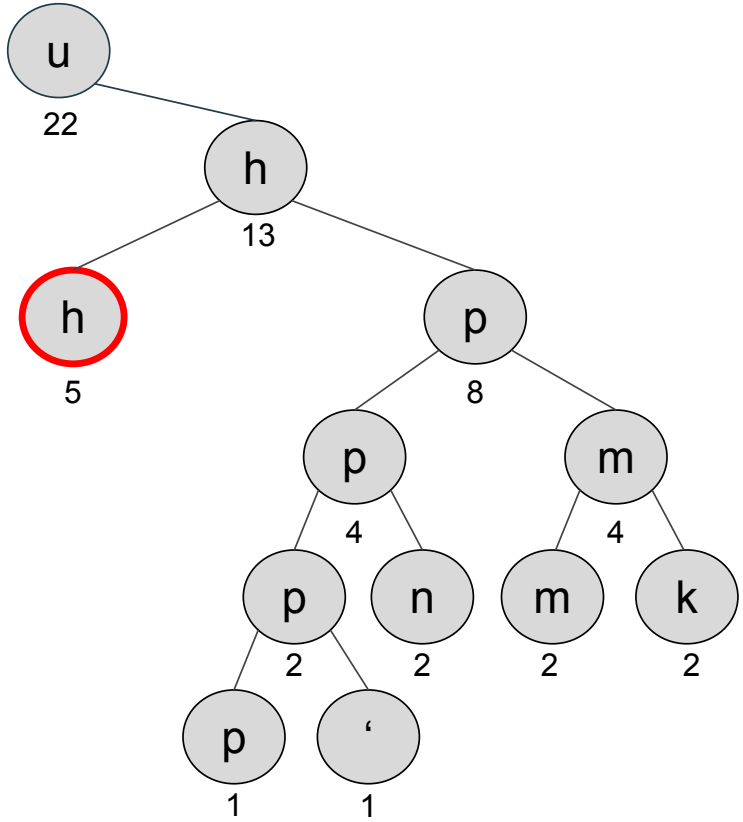

u
22
h
13
h
5
p
8
h
2
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

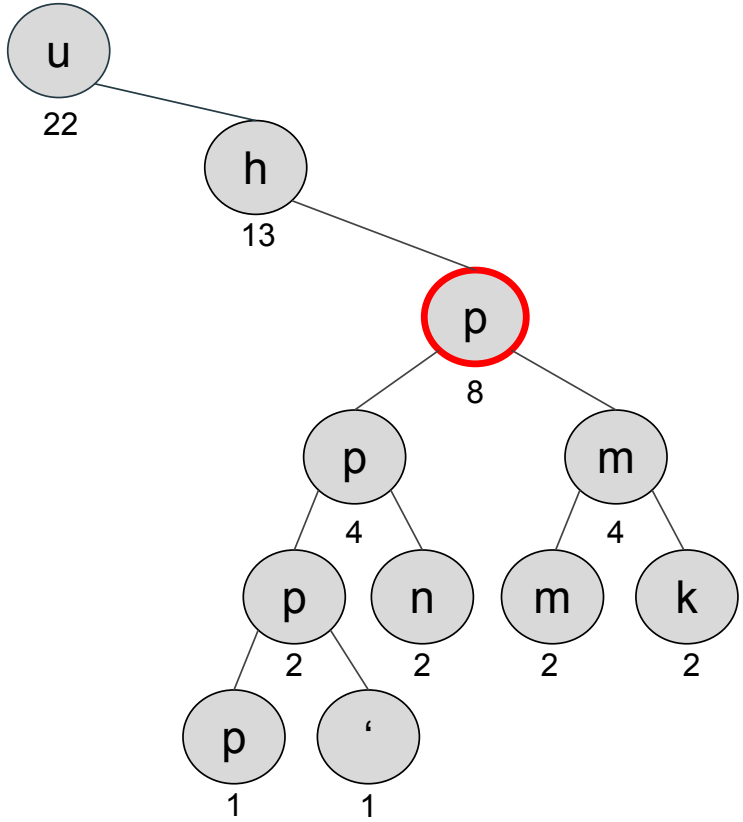

u
22
h
13
h
5
p
8
h
2
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

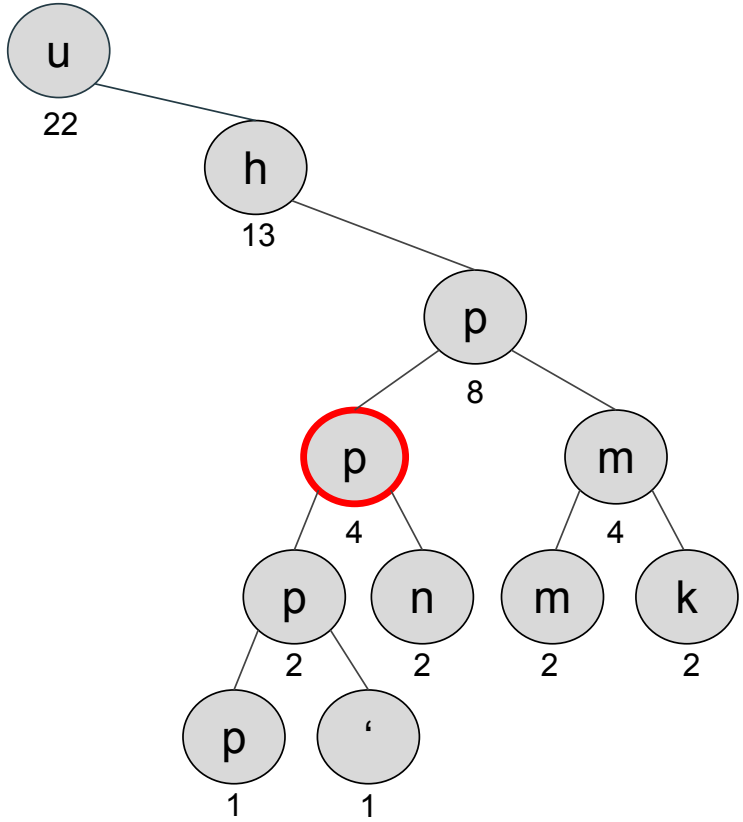

u
22
h
13
h
5
p
8
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

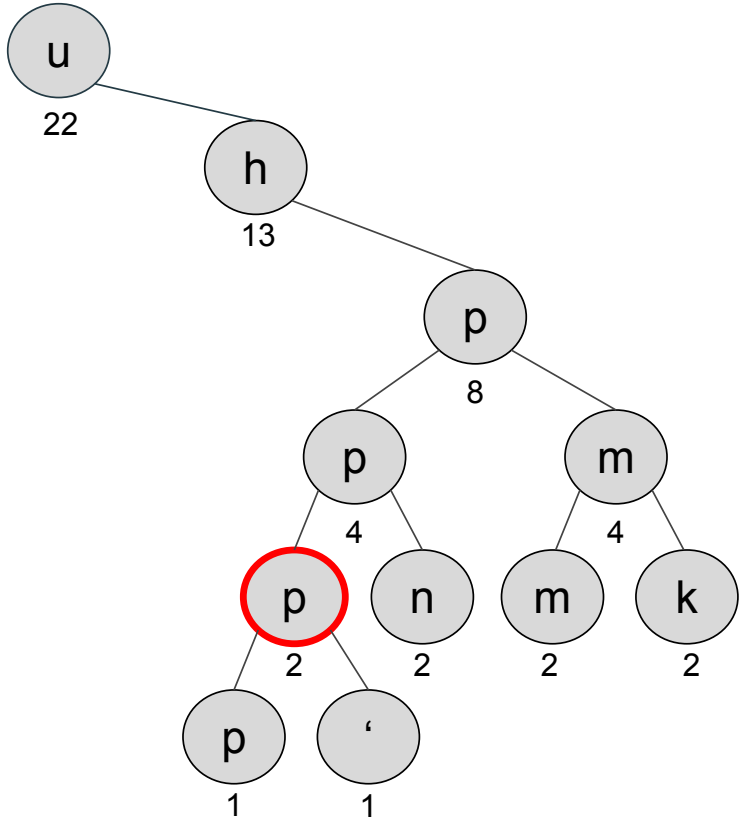

u
22
h
13
h
5
p
8
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

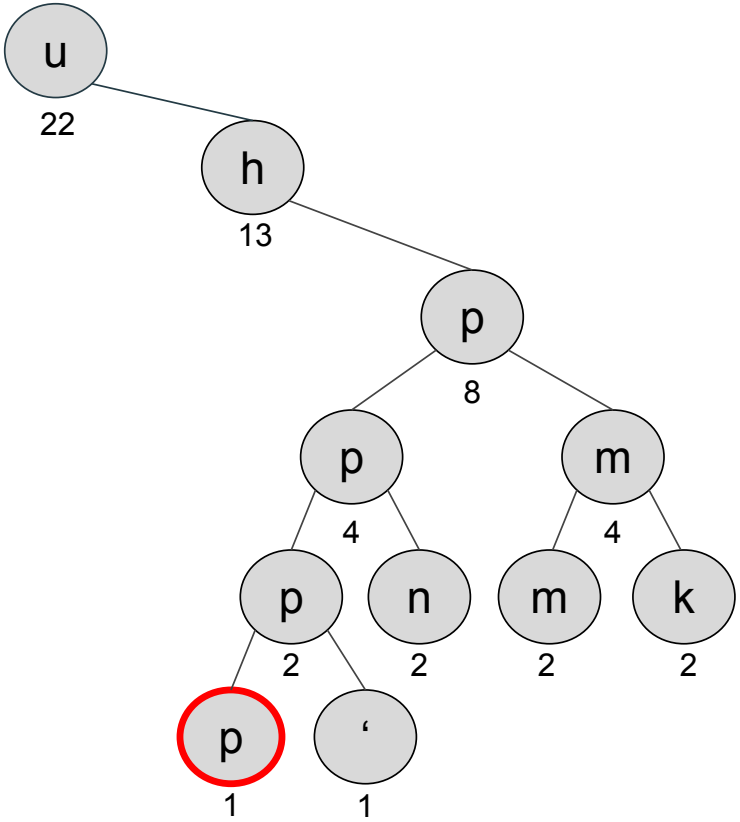

u
22
h
13
h
5
p
8
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

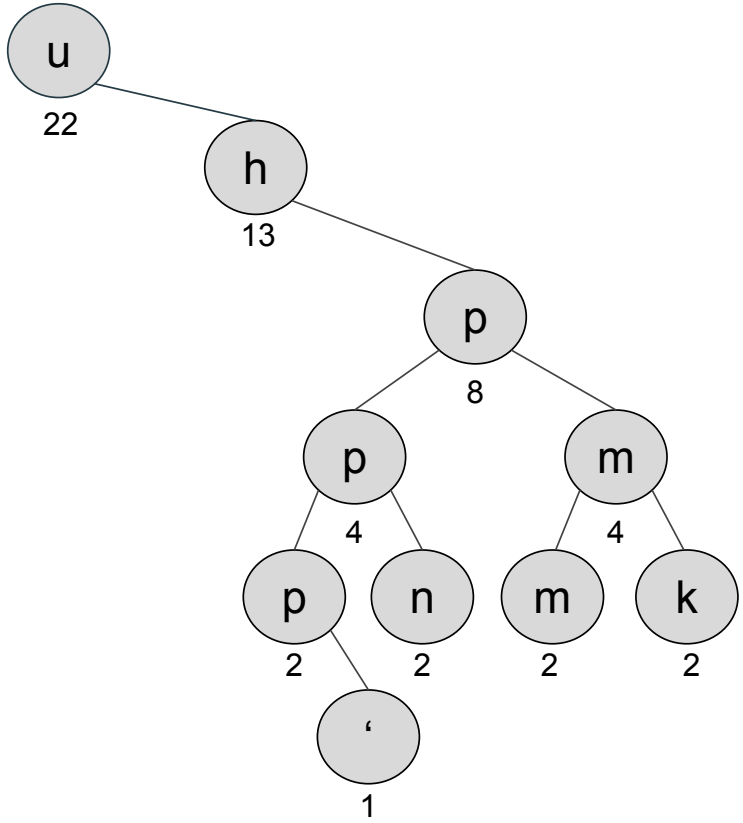

u
22
h
13
h
5
p
8
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

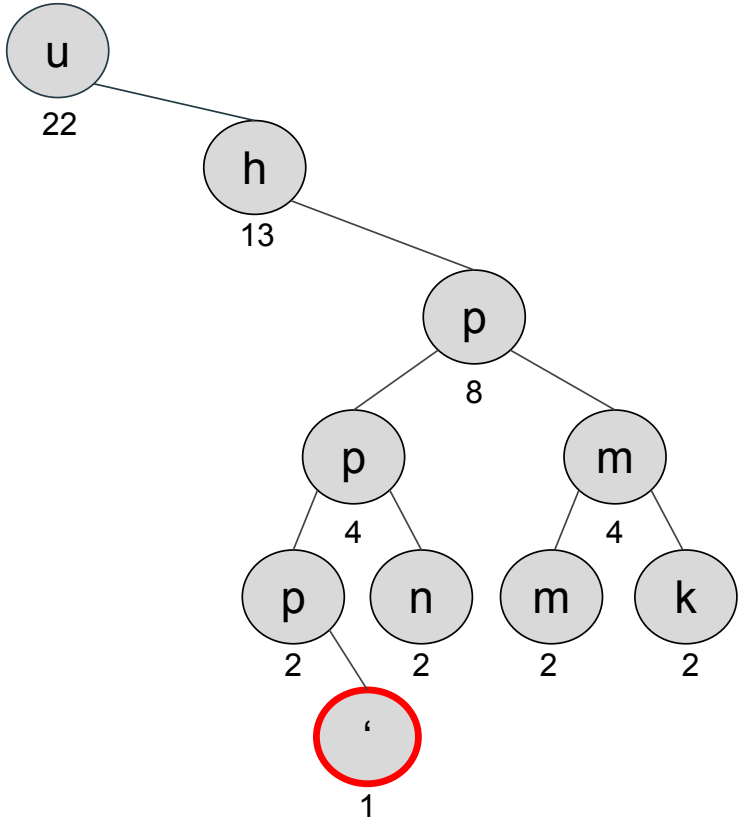

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

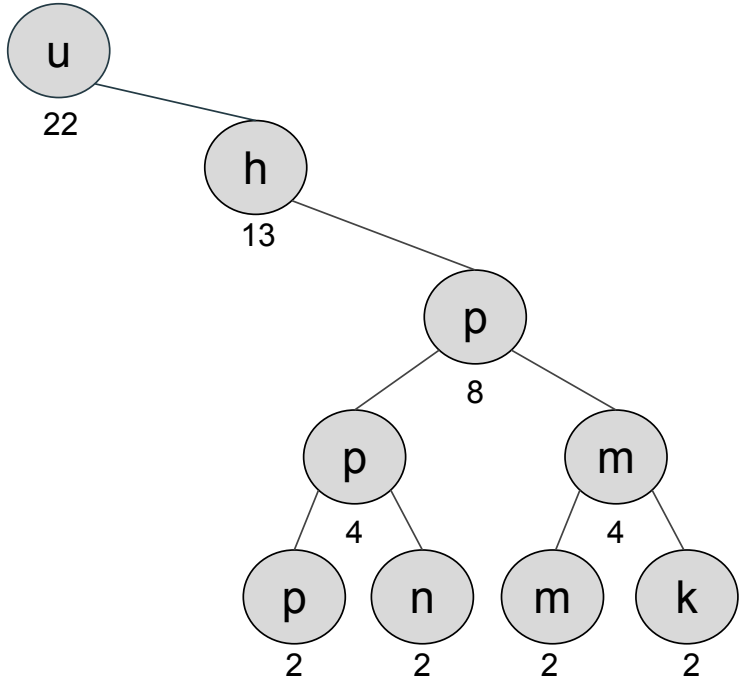

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

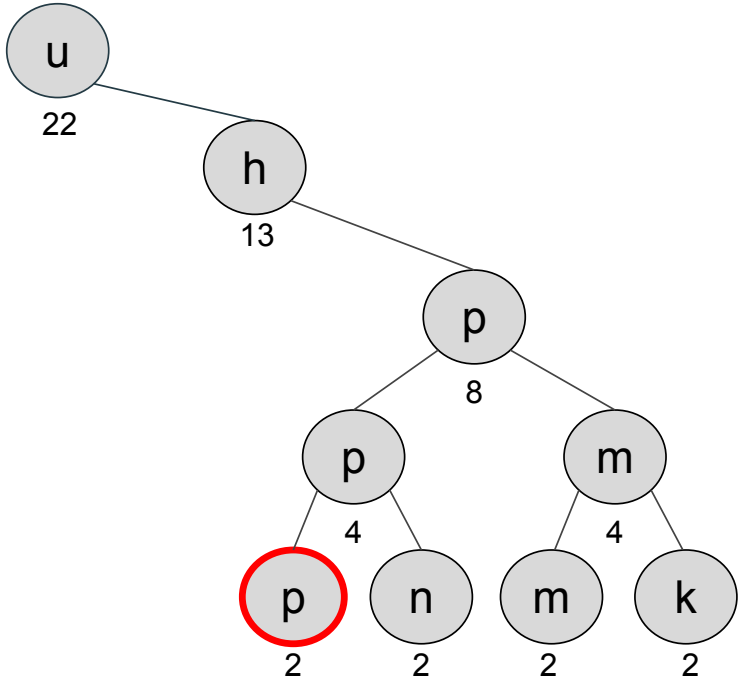

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

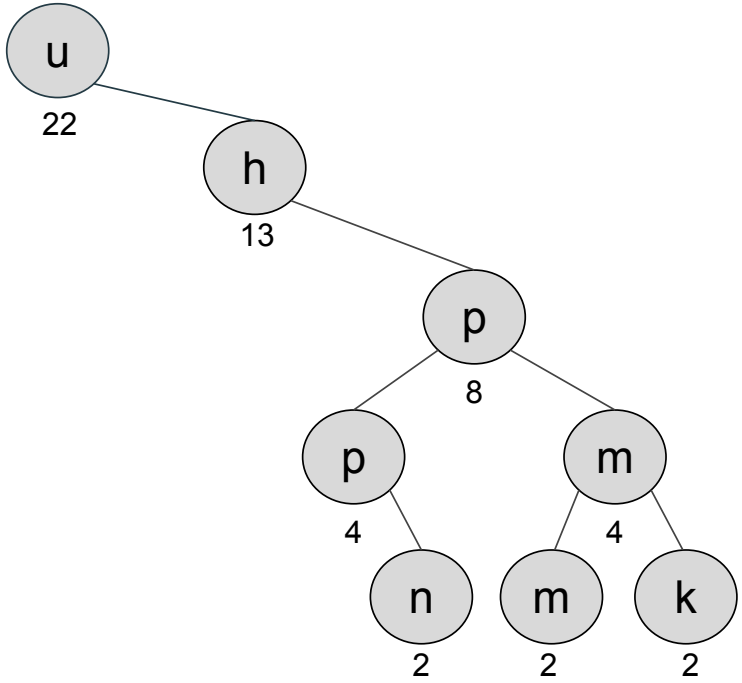

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1

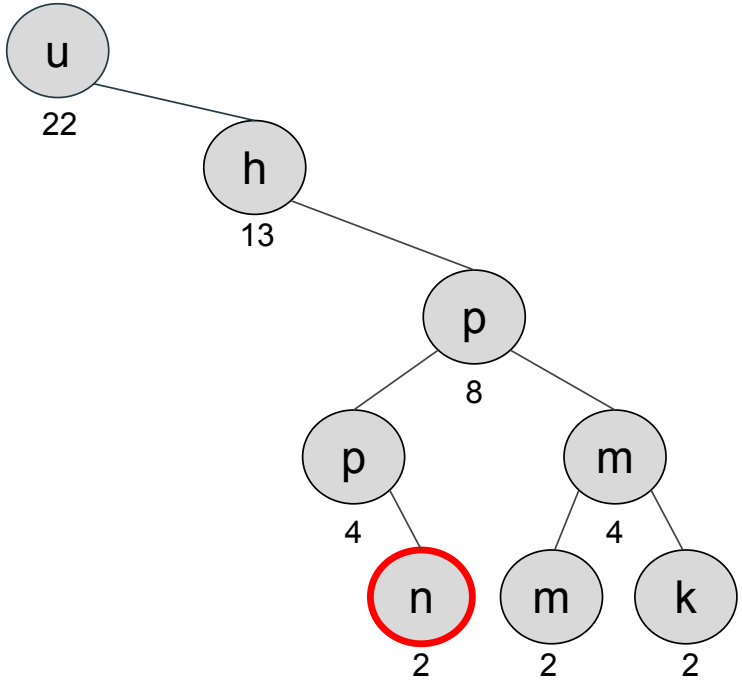

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2
‘
1

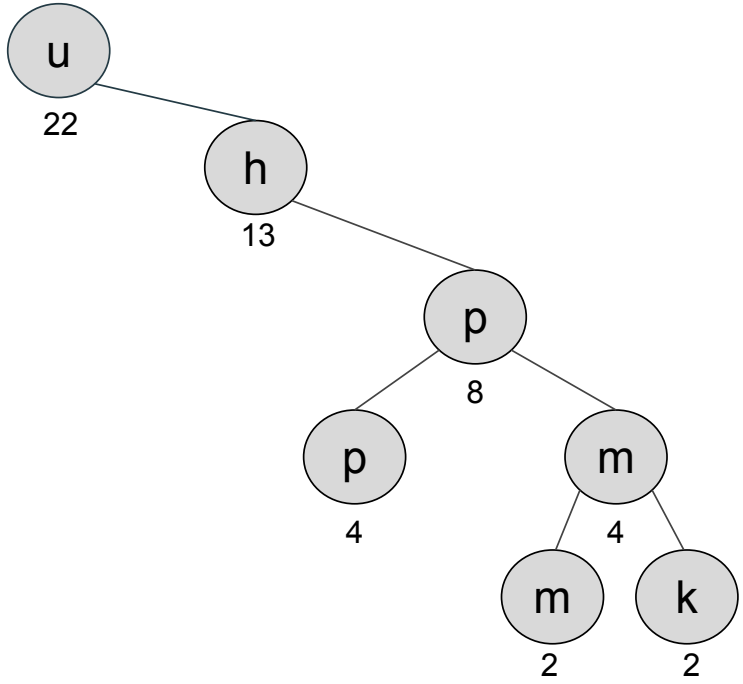

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2
‘
1

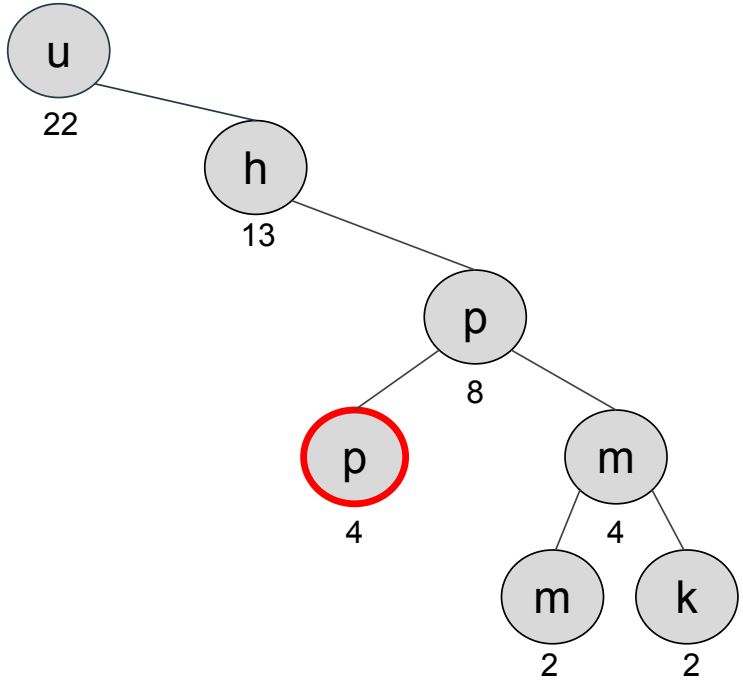

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2

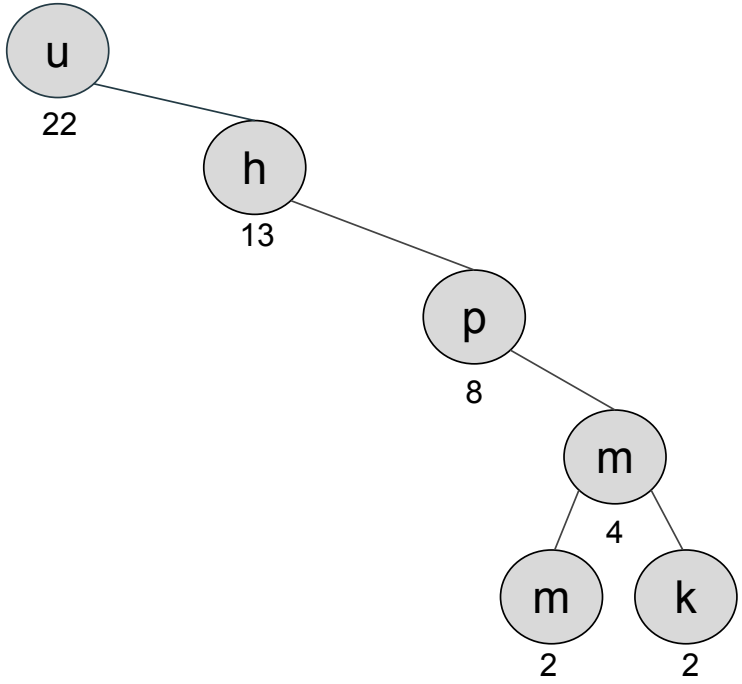

u
22
h
13
p
8
p
4
m
4
p
2
n
2
m
2
k
2

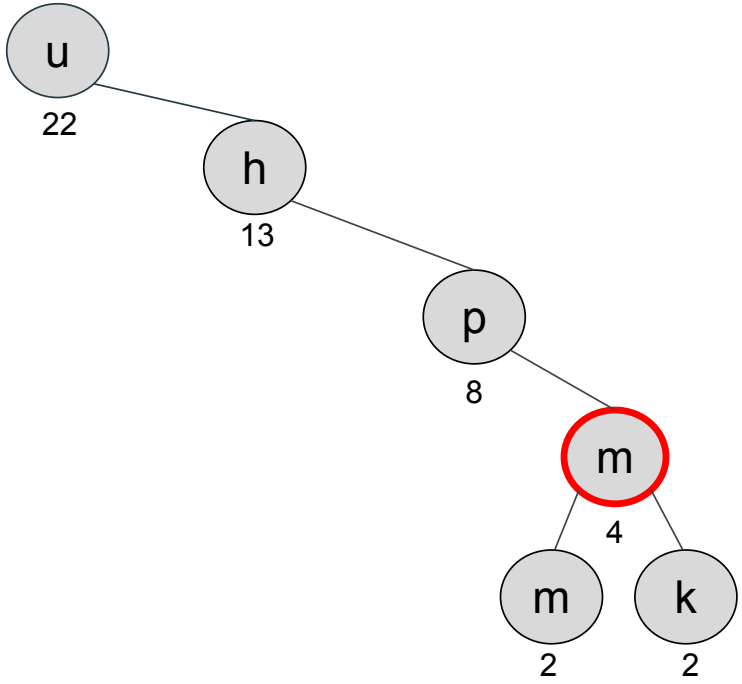

u
22
h
13
p
8
p
4
m
4
n
2
m
2
k
2

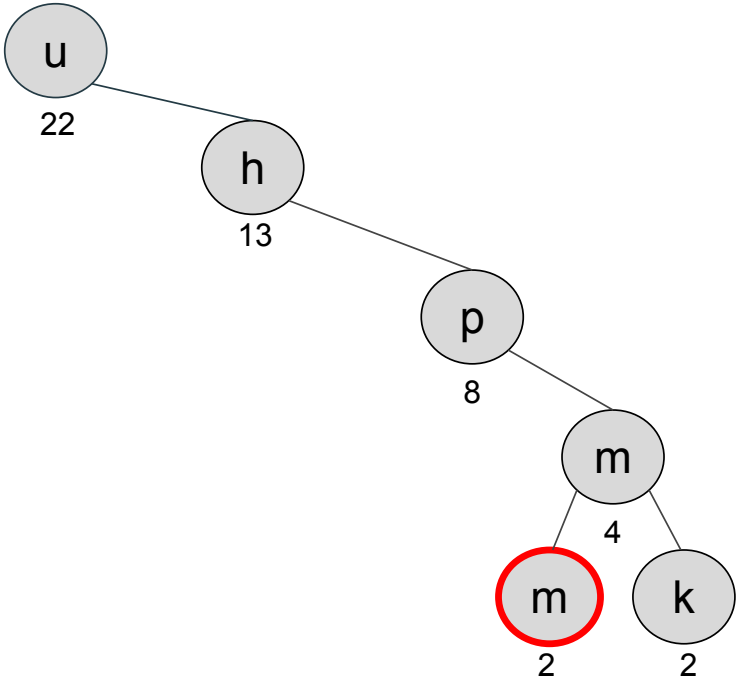

u
22
h
13
p
8
p
4
m
4
n
2
m
2
k
2

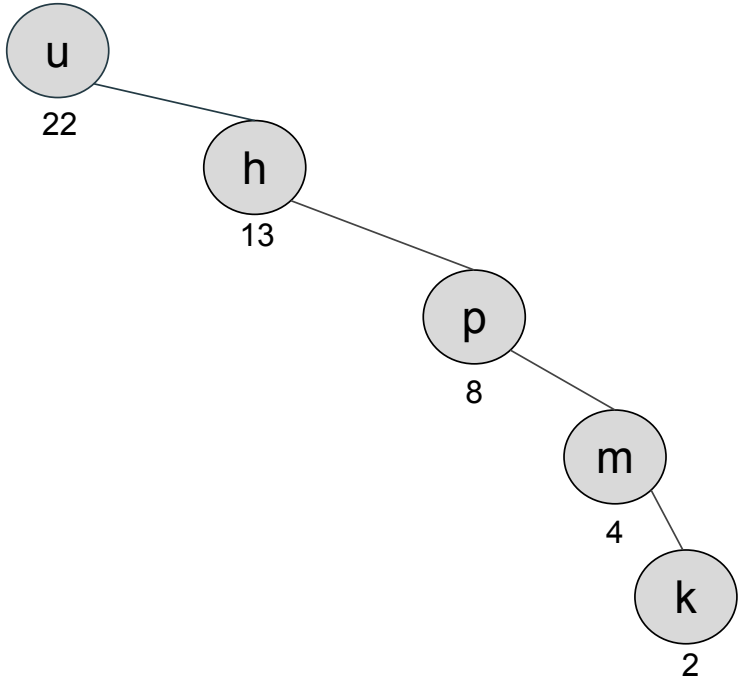

u
22
h
13
p
8
p
4
m
4
m
2
k
2

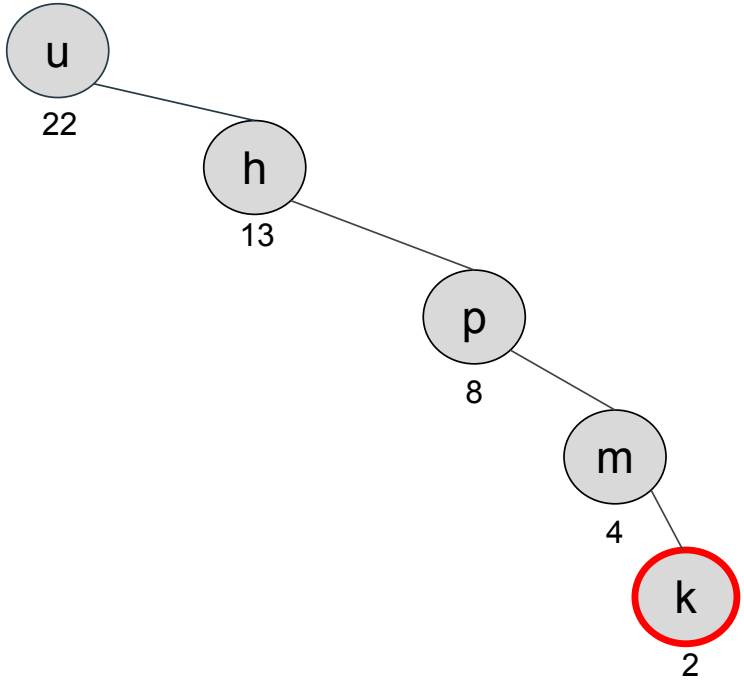

u
22
h
13
p
8
p
4
m
4
m
2
k
2

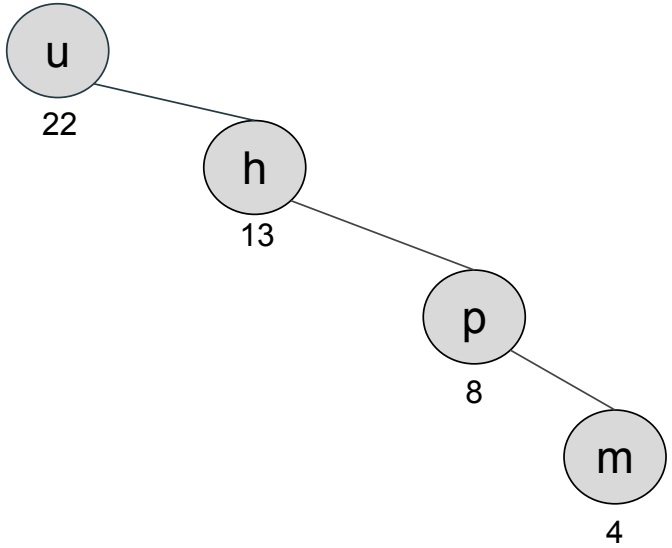

u
22
h
13
p
8
m
4
m
2
k
2

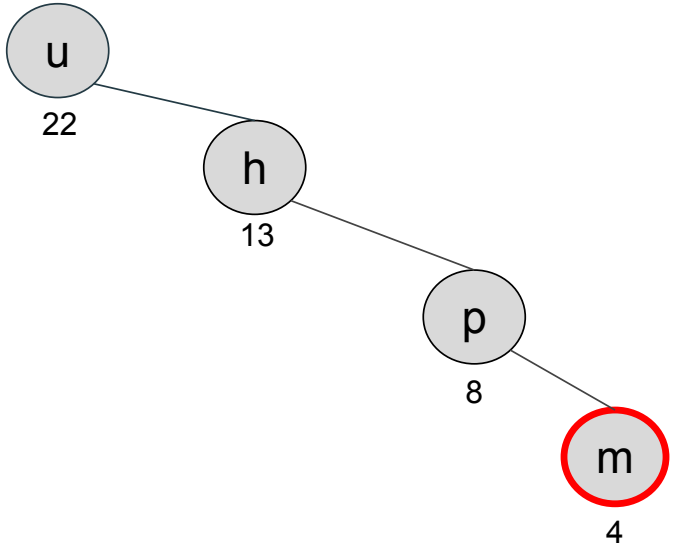

u
22
h
13
p
8
m
4
m
2
k
2

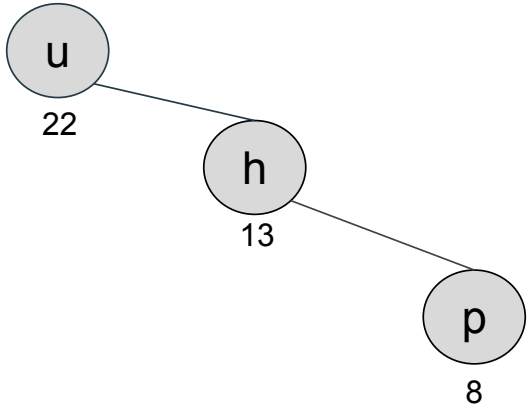

u
22
h
13
p
8
m
4
m
2
k
2

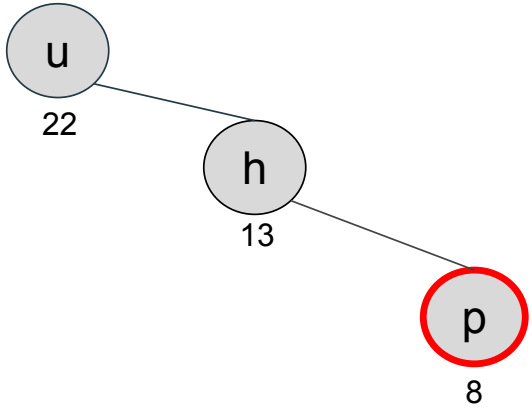

u
22
h
13
p
8
m
4
k
2

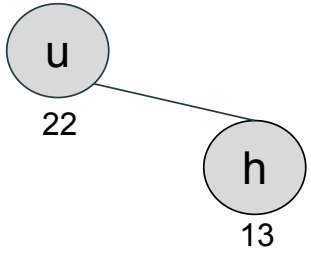

u
22
h
13
p
8
m
4
k
2

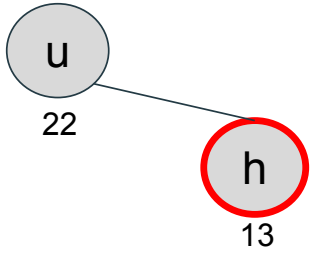

u
22
h
13
p
8
m
4


u
22
h
13
p
8
m
4

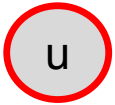

u
22
h
13
p
8


u
22
h
13
p
8


u
22
h
13


u
22
h
13

22



22

Done!

Building Header (Naive)
Encode & Compress

Encoding (pseudoCompression & encode)

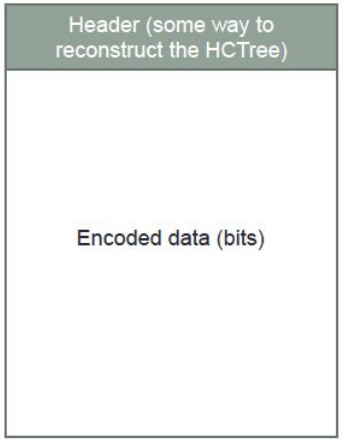
1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter)
4. Create new compressed file by saving **some information or representation of the huffman encoding tree** at the top of the file followed by the code for each symbol (letter) in the file

Encoding

1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter)
4. Create new compressed file by saving **the entire frequency set of the 256 ASCII characters at the top of the file** followed by the code for each symbol (letter) in the file

Writing the Compressed File

In order for your *uncompress* program to know what the Huffman Tree you use look like, you need to store some information at the top of your compressed file to let *uncompress* program rebuild the HCTree.



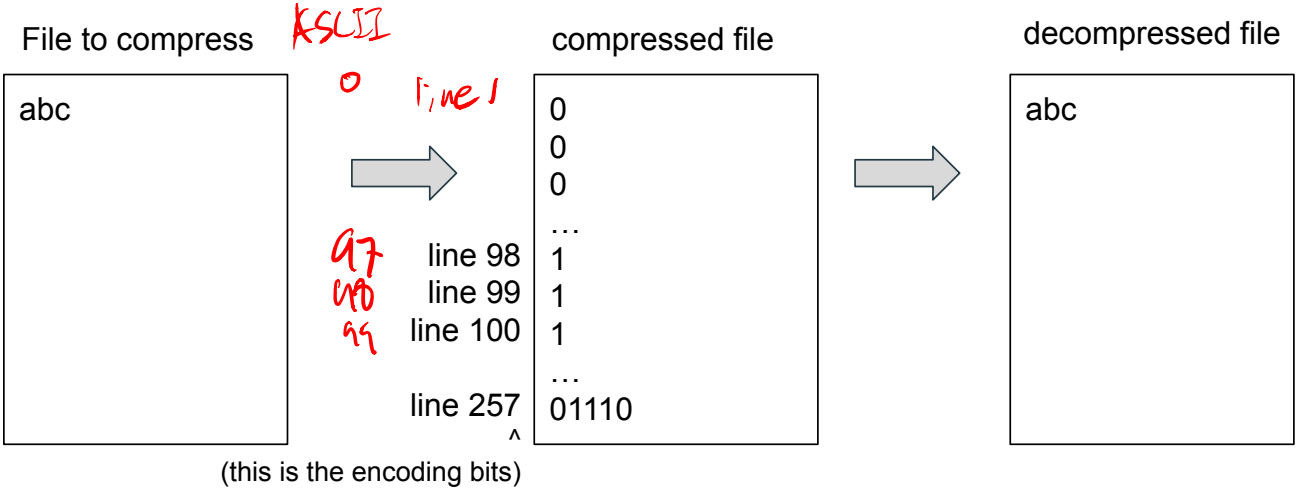
Store the ASCII char counts as the header

You need to store some information at the top of your compressed file to let *uncompress* program rebuild the HCTree.

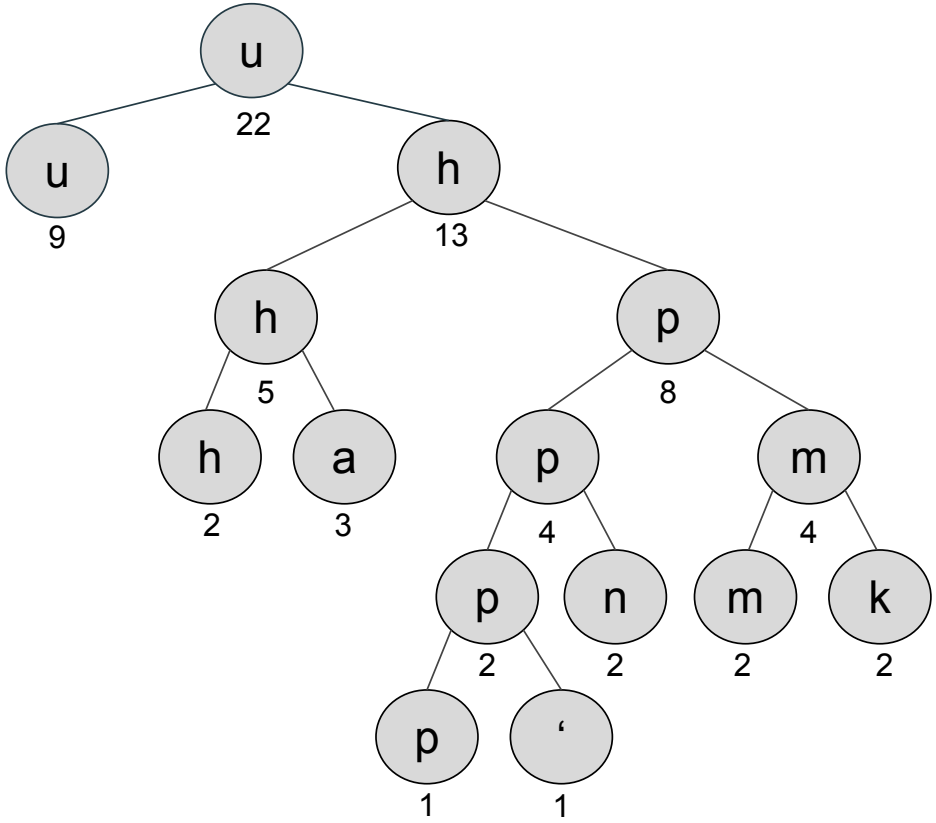
In order to re-use the HCTree::build(), we will just store the frequency of 256 ASCII symbols in the header of the file. i.e.:

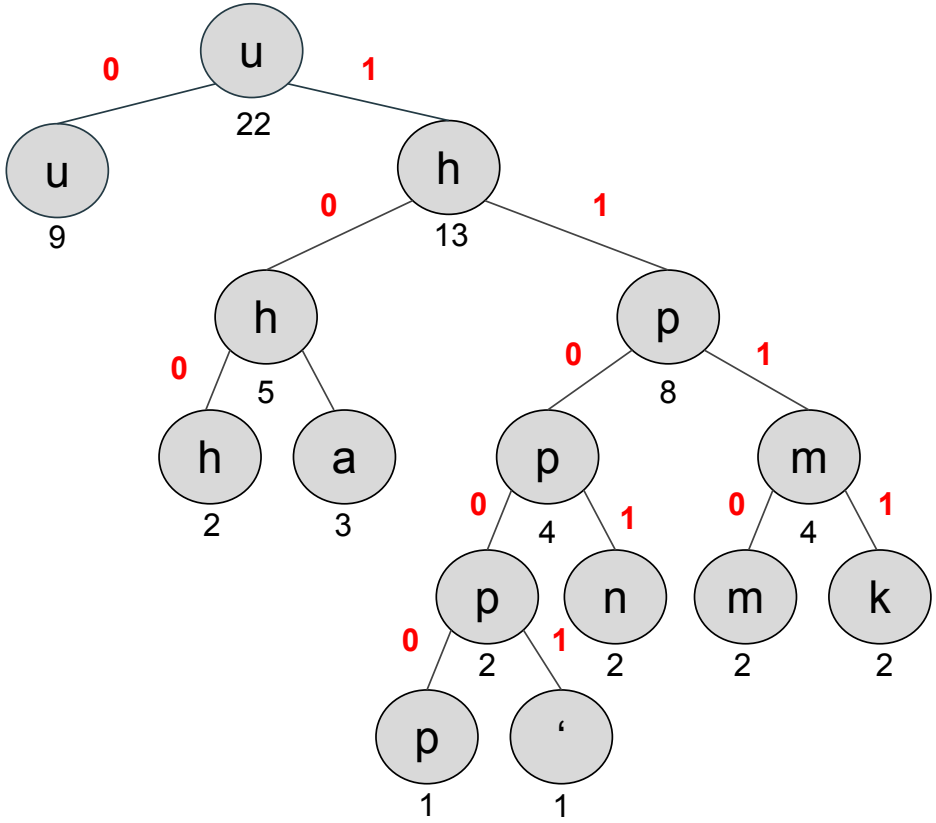
line 1 will list the count of ASCII 0 (NUL)
...
line 98 will list the count of ASCII 97 (‘a’)
…
line 257 will list the bit encoded data

Encoded File Example



Note: you can run the reference solution to get a better understanding of what compress.cpp should produce.

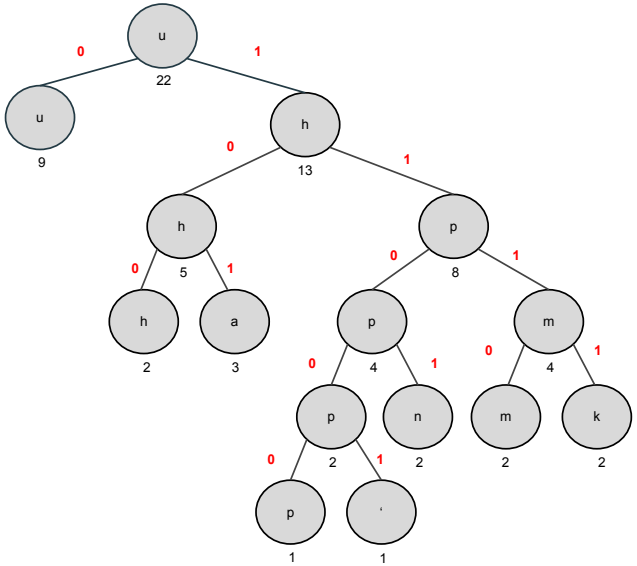

u
22
u
9
h
13
h
5
p
8
h
2
a
3
p
4
m
4
p
2
n
2
m
2
k
2
p
1
‘
1


u
22
0
1
u
9
h
13
0
1
h
5
p
8
0
1
h
2
a
3
p
4
m
4
0
1
0
1
p
2
n
2
m
2
k
2
0
1
p
1
‘
1

Encode the file

Input Message:

humuhumunukunukuapua’a



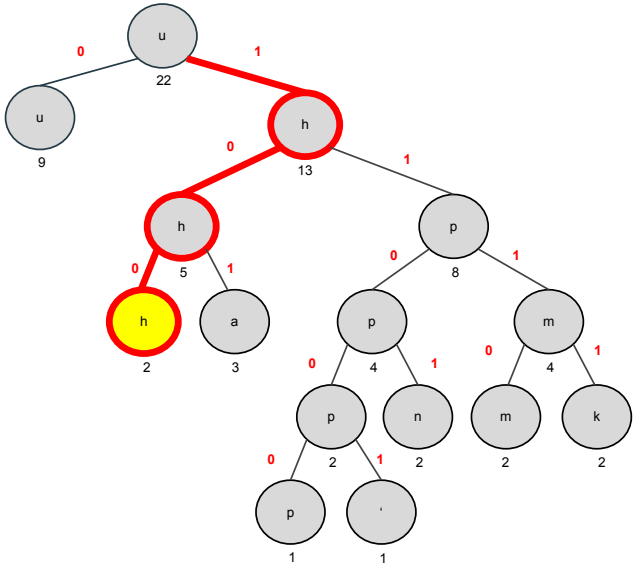
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

100



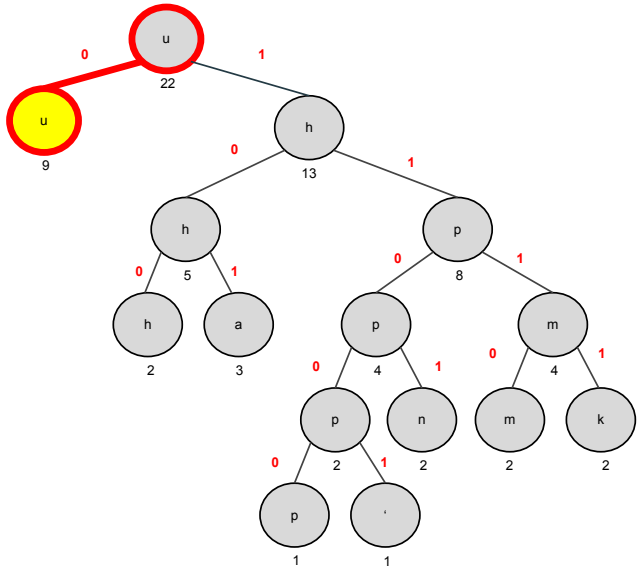
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

1000



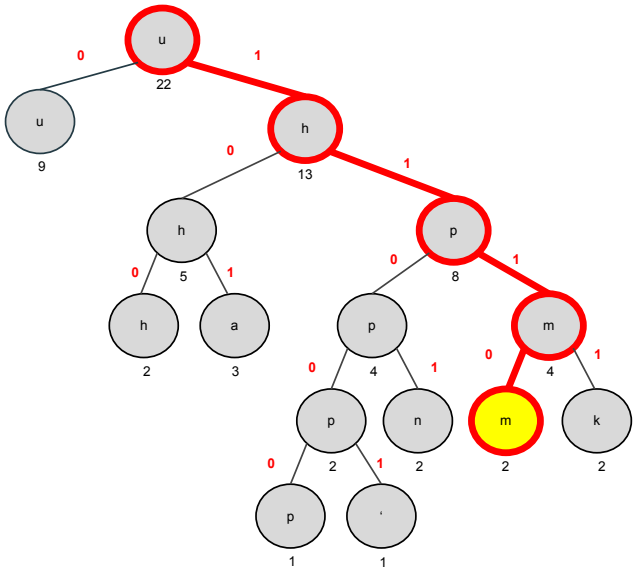
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

10001110



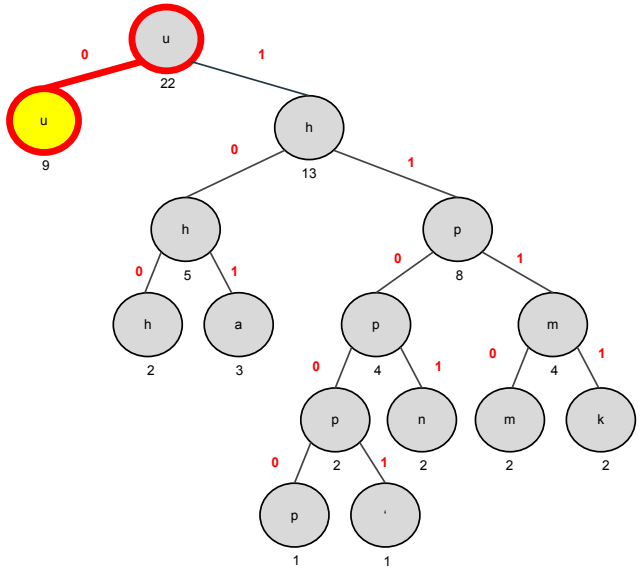
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

100011100



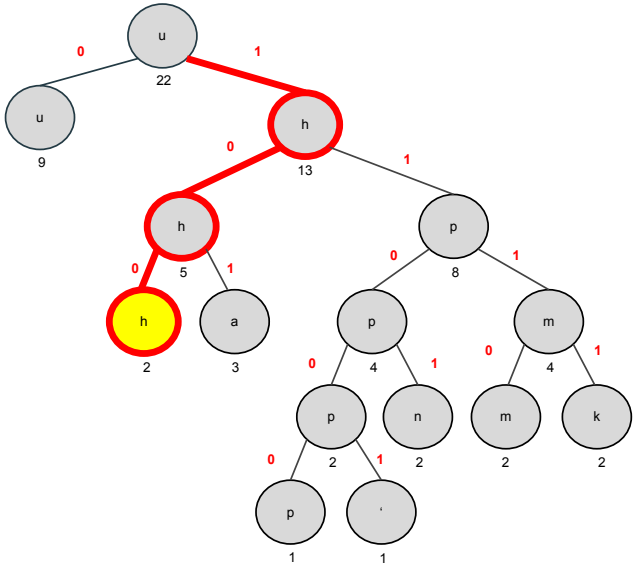
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

100011100100



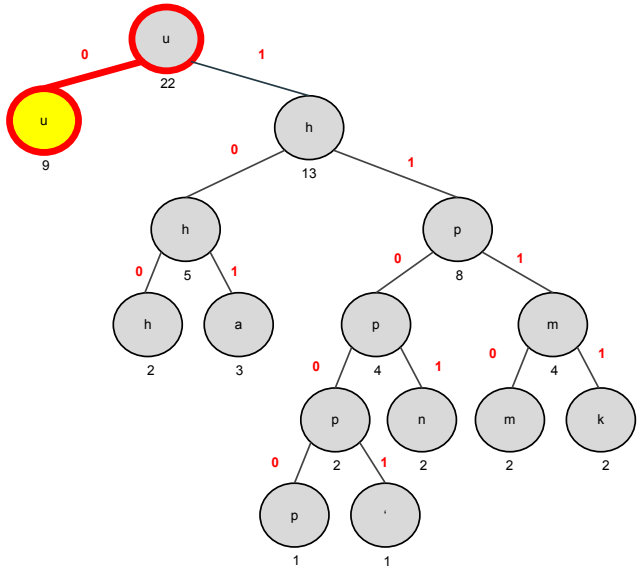
Encode the file

Input Message:

humuh**u**munukunukuapua’a

Output Message:

1000111001000



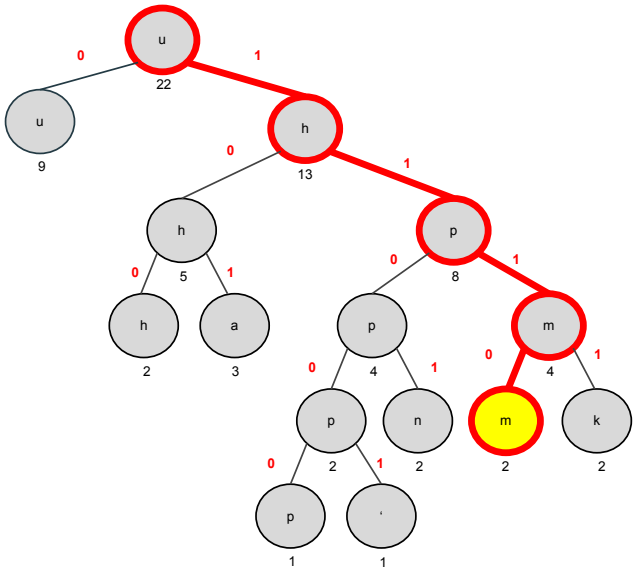
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

10001110010001110



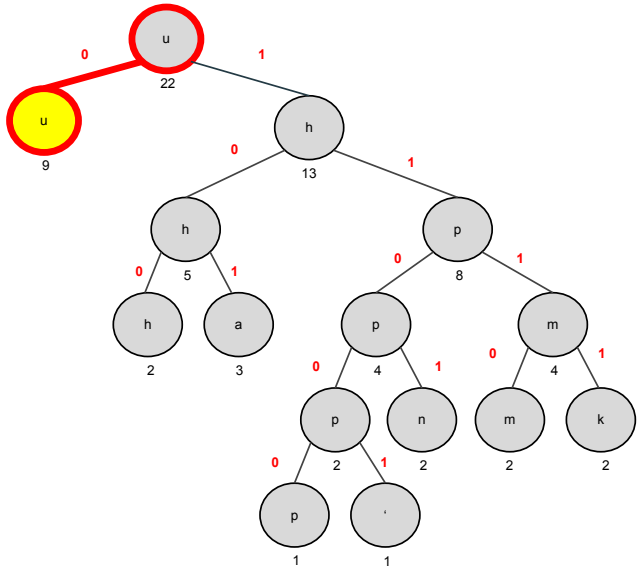
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

1000111001000111100



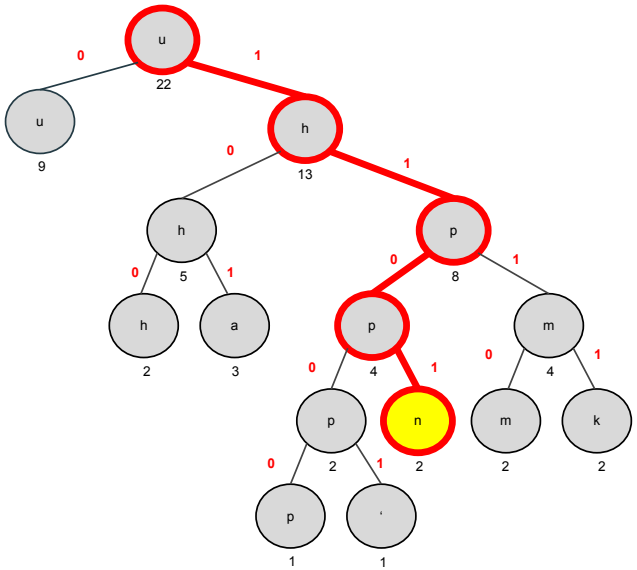
Encode the file

Input Message:

humuhumu**n**ukunukuapua’a

Output Message:

1000111001000111001**1101**



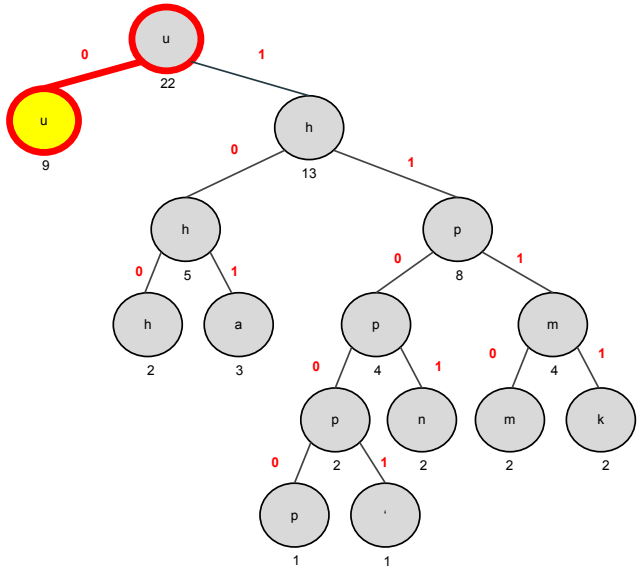
Encode the file

Input Message:

humuhumun**u**kunukuapua’a

Output Message:

1000111001000111001101**0**



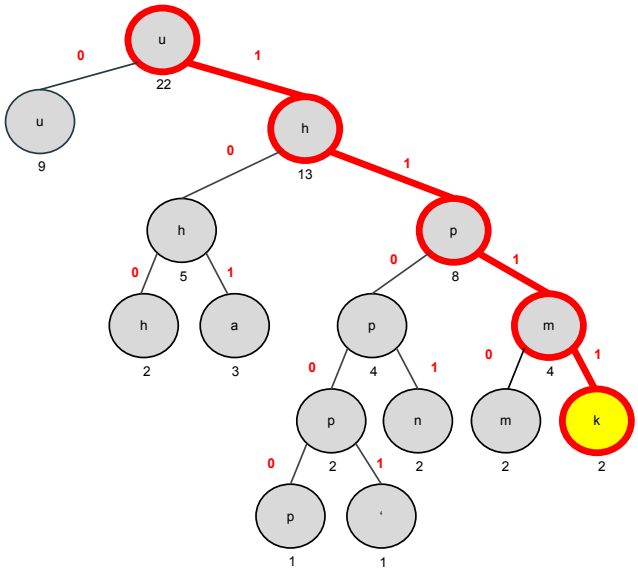
Encode the file

Input Message:

humuhumunu**k**unukuapua’a

Output Message:

1000111001000111001101011111



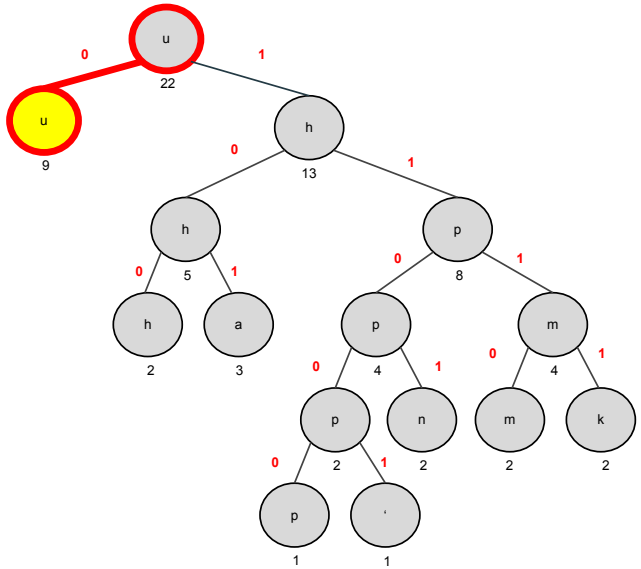
Encode the file

Input Message:

humuhumunuk**u**nukuapua’a

Output Message:

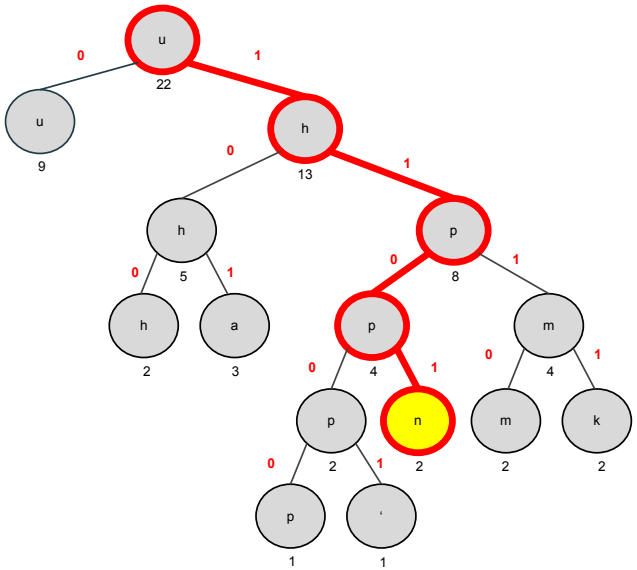
1000111001000111001101011110



Encode the file

Input Message:

humuhumunukunukuapua’a



Output Message:

1000111001000111001101011110**1101**

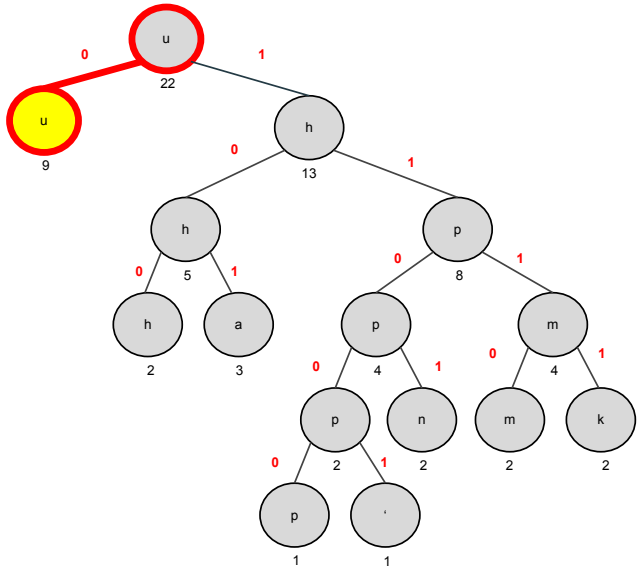
Encode the file

Input Message:

humuhumunukunukuapua’a

Output Message:

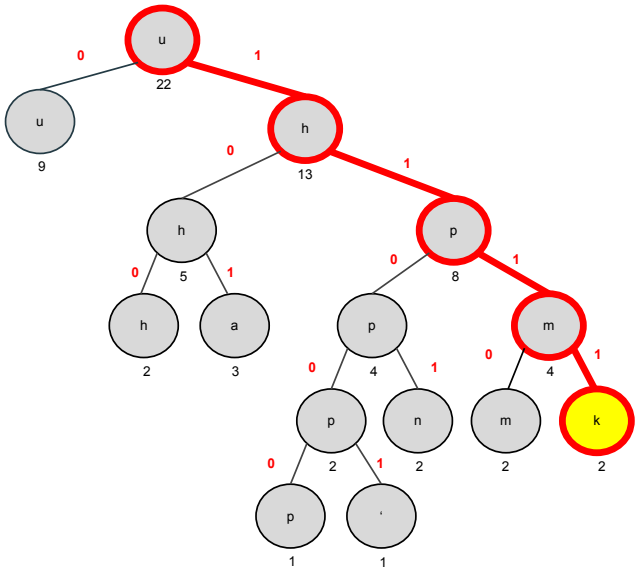
1000111001000111001101011110110 10



Encode the file

Input Message:

humuhumunukunu**k**uapua’a



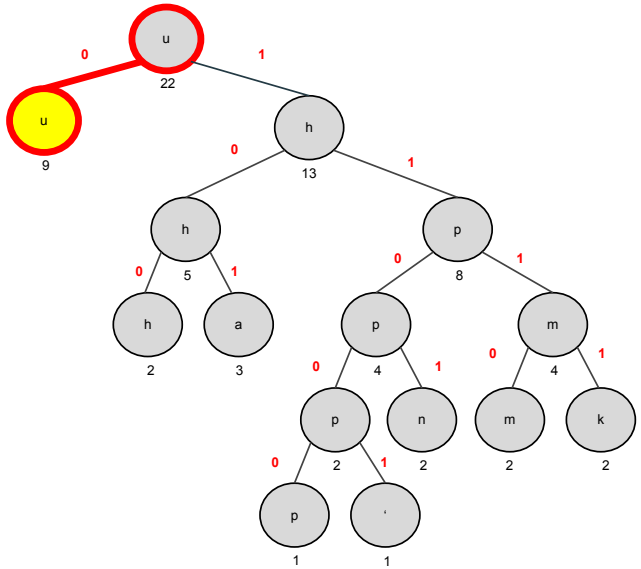
Output Message:

1000111001000111001101011110110101**1111**

Encode the file

Input Message:

humuhumunukunuk**u**apua’a



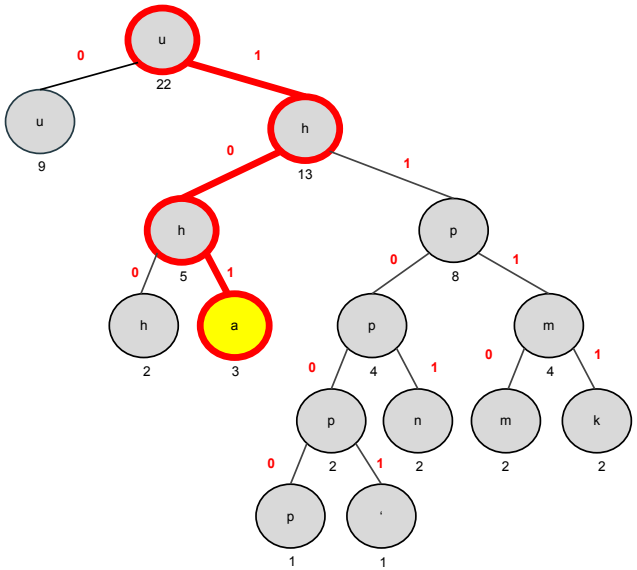
Output Message:

1000111001000111001101011110110101111**0**

Encode the file

Input Message:

humuhumunukunukuapua’a



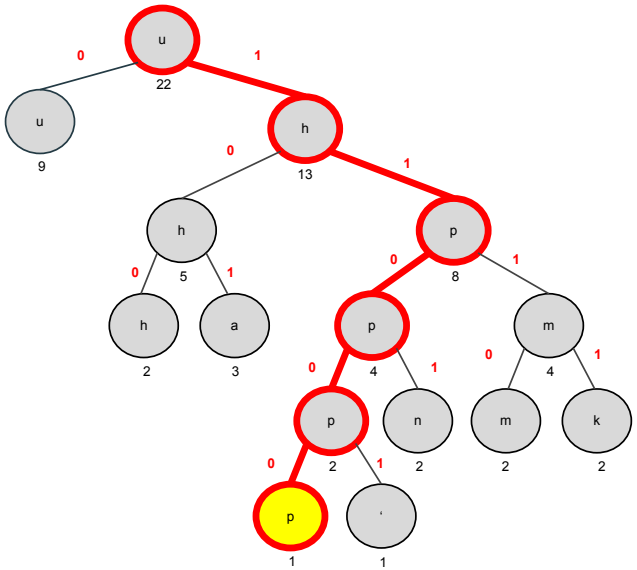
Output Message:

1000111001000111001101011110110101111010**101**

Encode the file

Input Message:

humuhumunukunukuapua’a



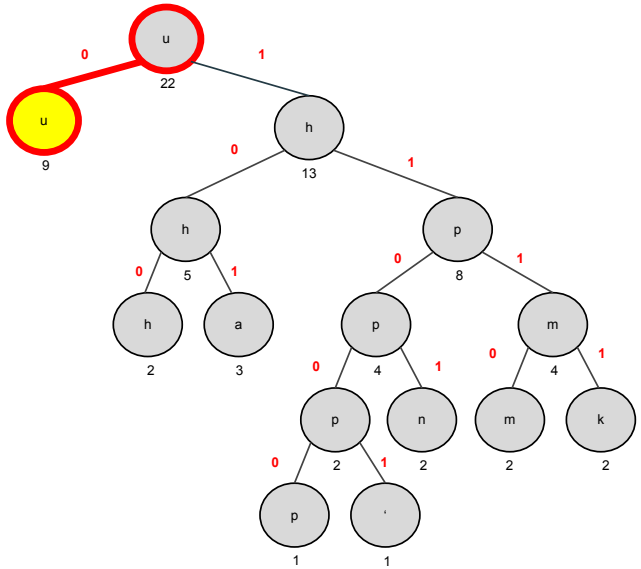
Output Message:

1000111001000111001101011110110101111010111000

Encode the file

Input Message:

humuhumunukunukuap**u**a’a



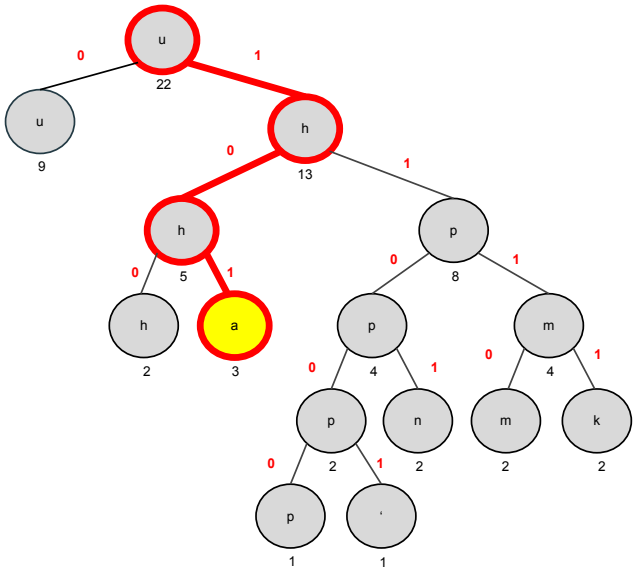
Output Message:

1000111001000111001101011110110101111010111000**0**

Encode the file

Input Message:

humuhumunukunukuapu**a**’a



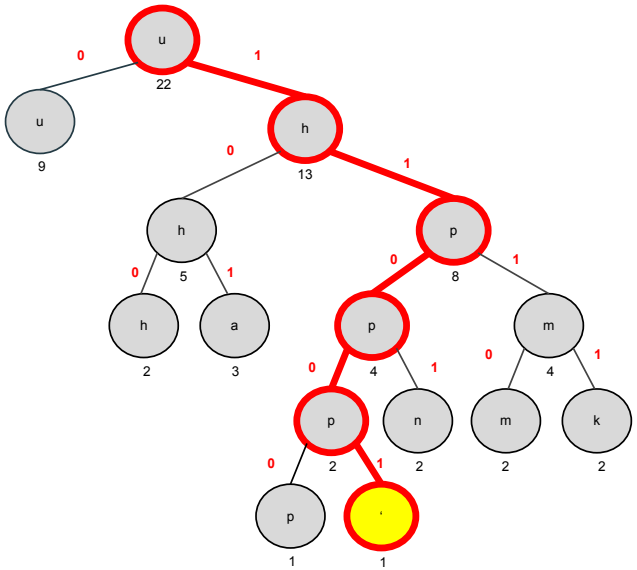
Output Message:

1000111001000111001101011110110101111010111000**0101**

Encode the file

Input Message:

humuhumunukunukuapua’a



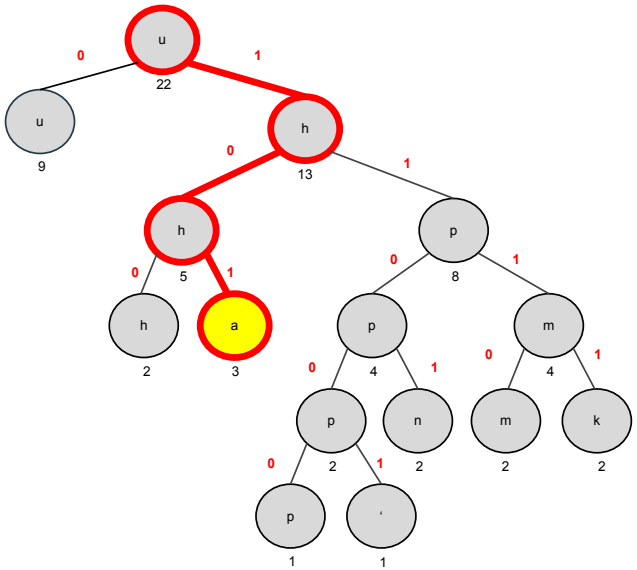
Output Message:

1000111001000111001101011110110101111010111000010111001

Encode the file

Input Message:

humuhumunukunukuapua’**a**



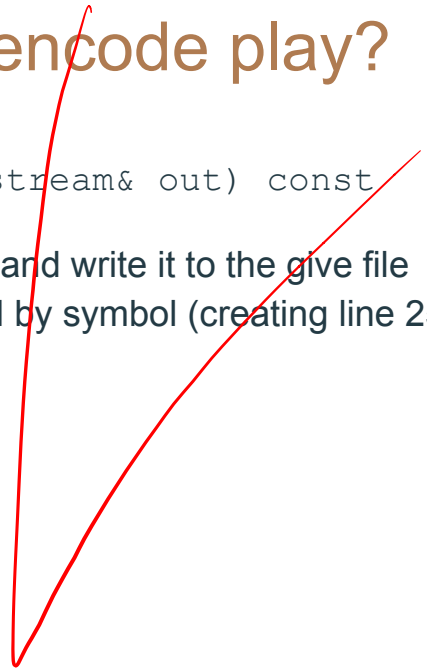
Output Message:

10001110010001110011010111101101011110101110000101110011**101**

Okay, so what role does encode play?

```
void HCTree::encode(byte symbol, ostream& out) const
```

- encode is used to encode a given symbol and write it to the give file
- it is used to build the encoded files symbol by symbol (creating line 257 of the naive header)



Okay, so how to we quickly encode our symbols?

well . . .

idk

Okay, so how do we quickly encode our symbols?

Remember we have `leaves`! (should have added all of the relevant `HCNode*` leaves to it in build)

This gives us access to all of the leaf nodes which corresponds to the symbols in our files and allows us to find the encoding of our leaf nodes in O(1)!

Decompressing Header (Naive)
Decode & Decompress

Decoding (pseudoDecompress, decode)

1. Read the file header (which contains the code) to recreate the tree
    - read frequencies of letters
    - build the tree

2. Decode each letter by reading the rest of the file and using the tree

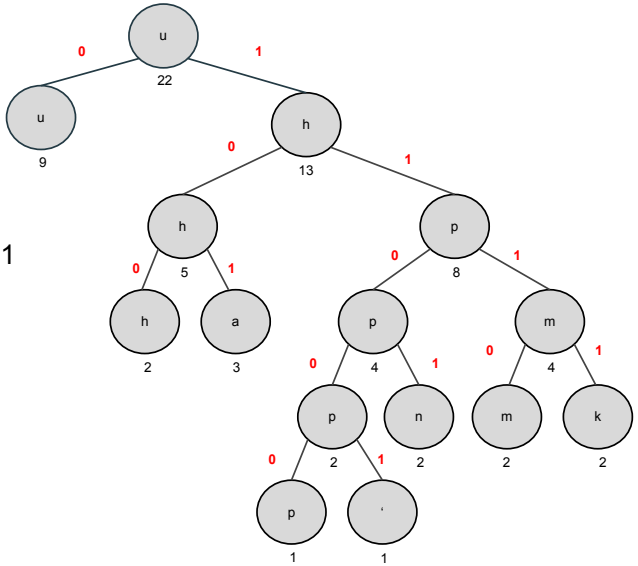
3. Write to the output file

Encode the file

Encoded Message:

10001110010001110011010111101101011110101110000101110011101

Decoded Message:

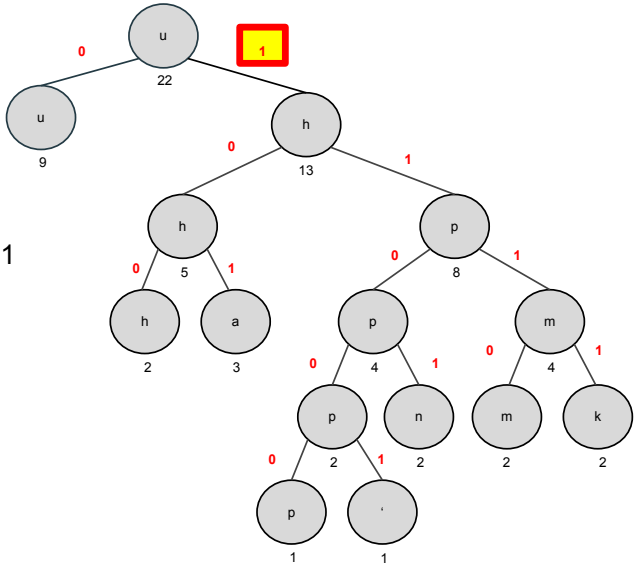


Encode the file

Encoded Message:

1000111001000111001101011110110101111010111000010111001101

Decoded Message:

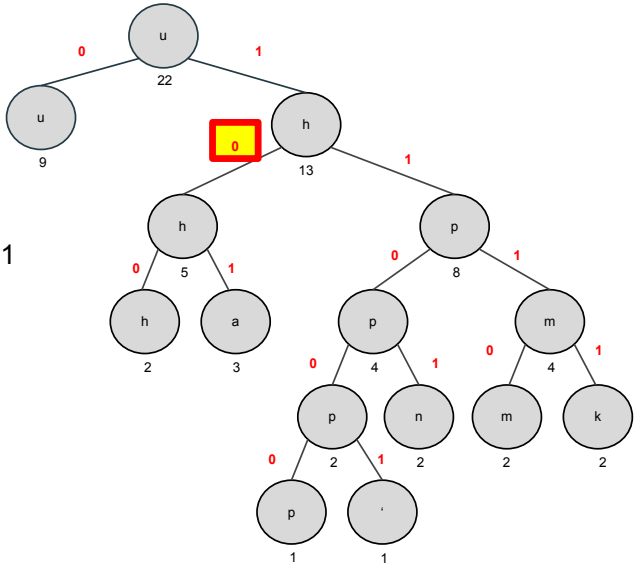


Encode the file

Encoded Message:

1000111001000111001101011110110101111010111000010111001101

Decoded Message:

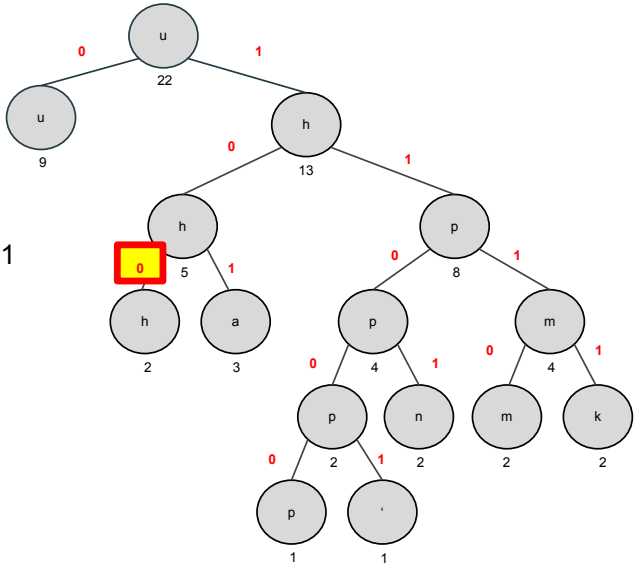


Encode the file

Encoded Message:

1000111001000111001101011110110101111010111000010111001101

Decoded Message:



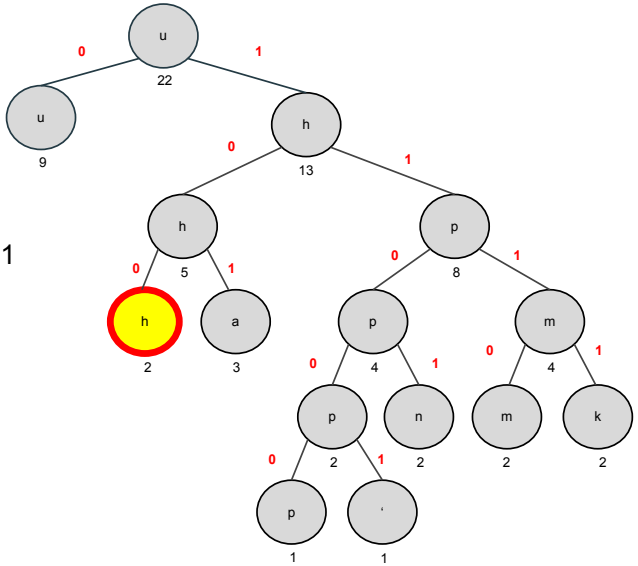
Encode the file

Encoded Message:

100011100100011100110101111011010111101011100001011100110 1

Decoded Message:

h



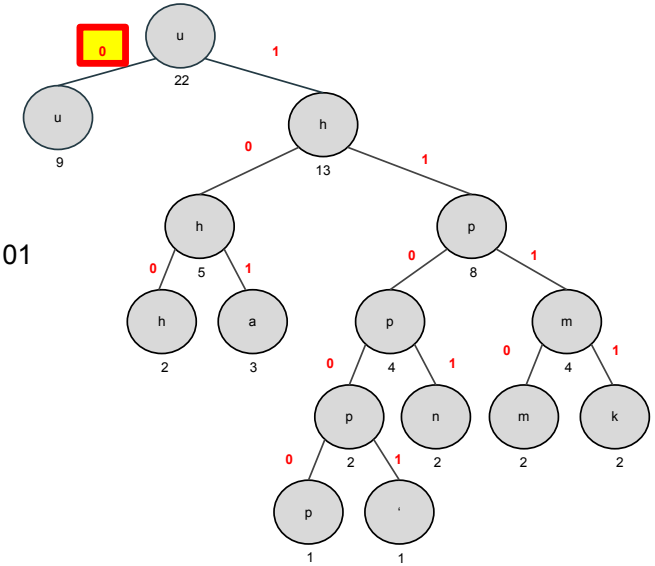
Encode the file

Encoded Message:

1000111001000111001101011110110101111010111000010111001101

Decoded Message:

h



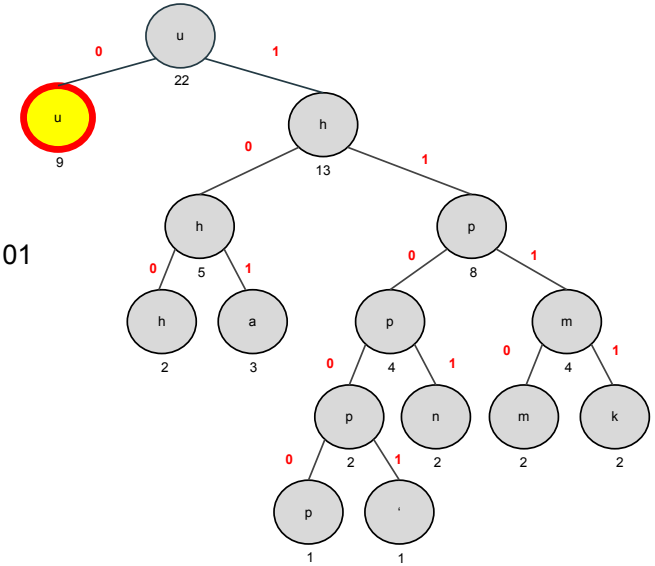
Encode the file

Encoded Message:

1000111001000111001101011110110101111010111000010111001101

Decoded Message:

hu



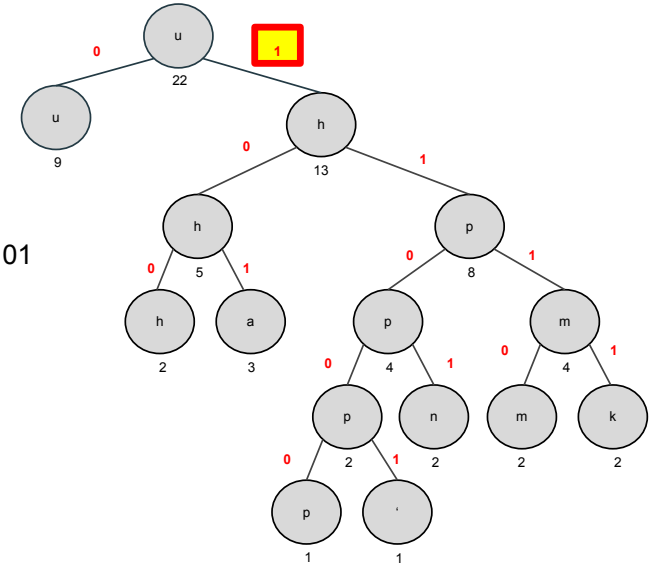
Encode the file

Encoded Message:

1000**1**1100100011100110101111011010111101011100001011100110**1**

Decoded Message:

hu



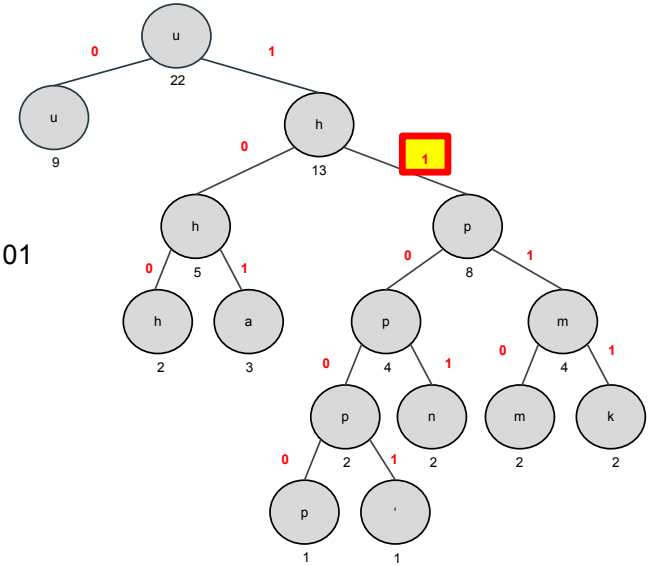
Encode the file

Encoded Message:

10001**1**1001000111001101011111011010111110101110000101110011101

Decoded Message:

hu



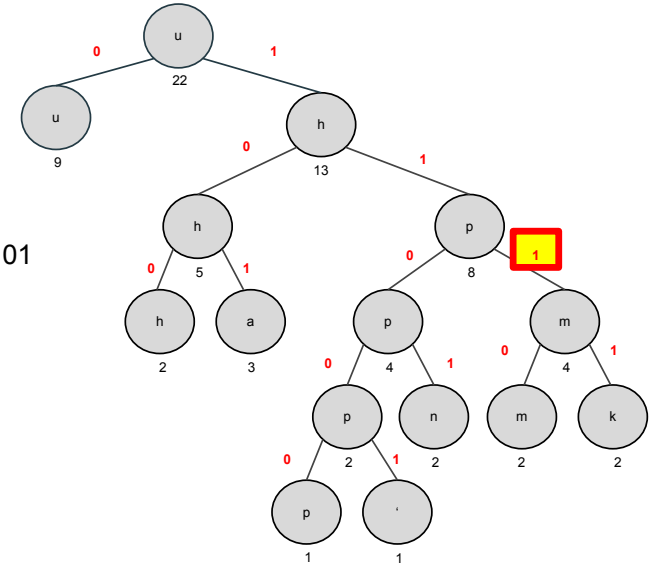
Encode the file

Encoded Message:

1000111001000111001101011110110101111010111000010111001101

Decoded Message:

hu



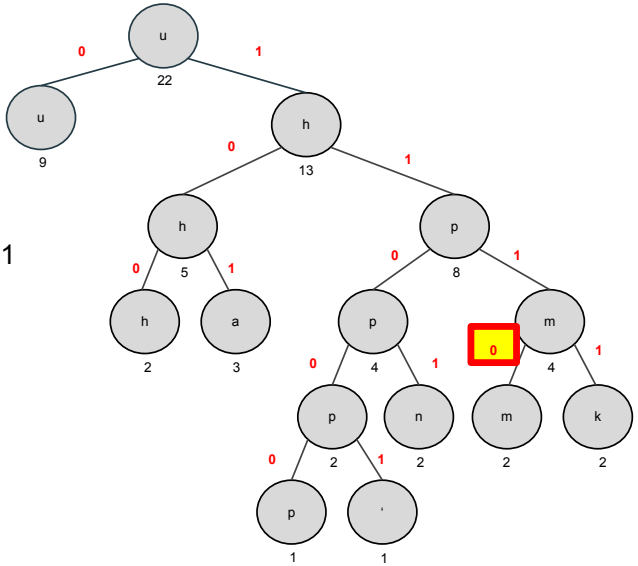
Encode the file

Encoded Message:

1000111**0**0100011100110101111011010111101011100001011100110 1

Decoded Message:

hu



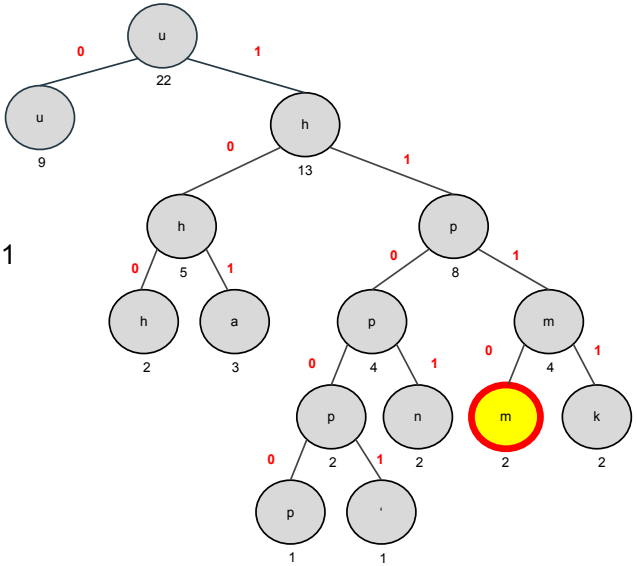
Encode the file

Encoded Message:

1000**1110**0100011100110101111011010111101011100001011100110 1

Decoded Message:

hum



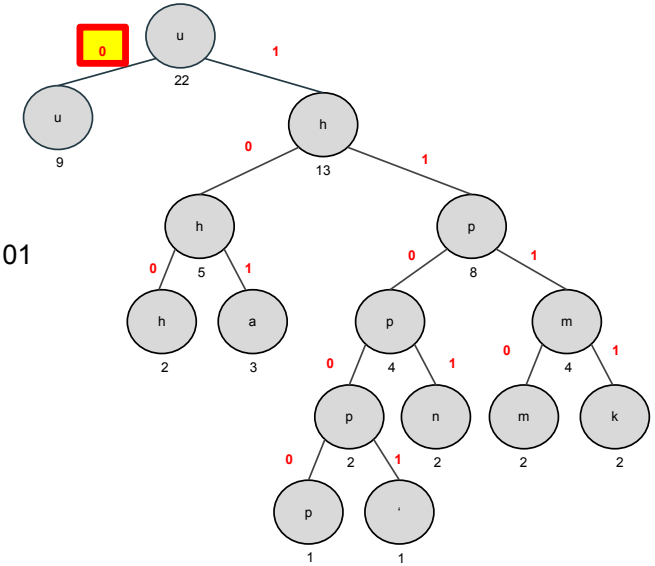
Encode the file

Encoded Message:

10001110**0**10001110011010111101101011110101110000101110011101

Decoded Message:

hum



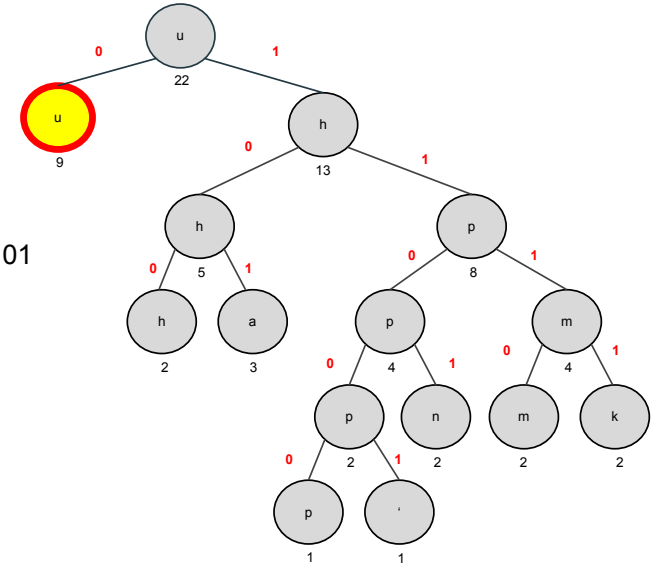
Encode the file

Encoded Message:

10001110**0**10001110011010111101101011110101110000101110011101

Decoded Message:

humu



Encode the file

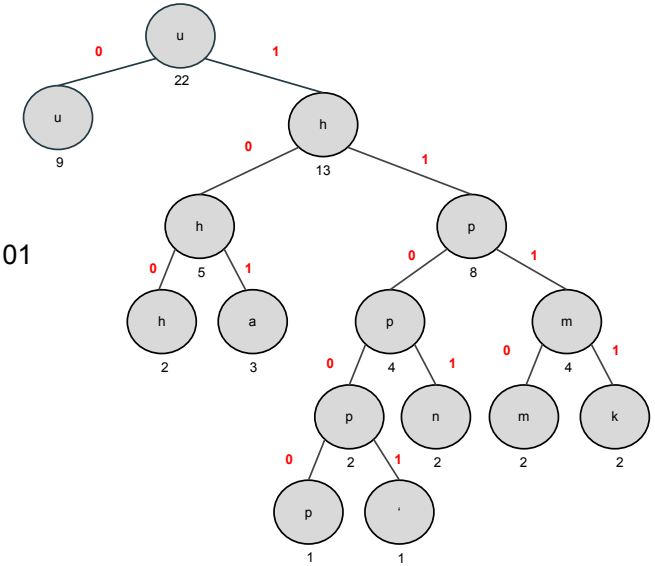
Encoded Message:

10001110010001110011010111101101011110101110000101110011101

Decoded Message:

humuhumunukunukuapua’a

Continues until you decode the entire message/file!



Okay, so what role does decode play?

```
byte HCTree::decode(istream& in) const
```

- decode is used to decode a given symbol from line 257
- it is used to decode the message/file
- In decode, you should keep reading from the input file until you read a leaf node. Decode will return once you reach a symbol, so read from the file until you reach a leaf.

CMakeList Files

CMake Intro & Demo

- CMake is an open-source, cross-platform family of tools designed to build, test and package software.
- CMake is used to control the software **compilation** process using simple platform and compiler independent configuration files, and generate native **makefiles** and workspaces that can be used in the compiler environment of your choice.

CMake: project()

The top-level CMakeLists.txt file for a project must contain a literal, direct call to the project() command

This sets the name of the **CMake** project

```
project(huffman_encoder)
```

CMake: add_subdirectory()

This adds a subdirectory to the build. The CMakeLists.txt file in the specified source directory will be processed immediately by **CMake** before processing in the current input file continues beyond this command.

```
add_subdirectory(test)
```

```
test
  ├── CMakeLists.txt
  └── test_HCTree.cpp
```

CMake: adding subprojects

In CMake, just use **`add_subdirectory()`** to add in **CMake** subprojects.

```
add_subdirectory(subprojects/googletest-release-1.8.1)
```

```
subprojects
    └── gtest
```

CMake: add_custom_target()

This adds a target with the given name that executes the given commands.

```
add_custom_target(cov COMMAND
${CMAKE_CURRENT_LIST_DIR}/build_scripts/generate_coverage_report
-r "make" "make test")
```

target_include_directories()

CMake specifies include directories or targets to use when compiling a given target. The named <target> must have been created by a command such as add_executable or add_library.

```
#include "HCTree.hpp"
```

```
encoder
    ├── HCNode.hpp
    ├── HCTree.hpp
    └── CMakeLists.txt
```

```
target_include_directories(huffman_encoder PUBLIC .)
```

Linking Dependencies

CMake links dependencies directly using functions:

```
target_include_directories()
target_link_libraries()
```

CMake: add_libraries()

You need to use this in order to be able to declare a dependency that does not only include .hpp files, but **also .cpp files**.

CMake:

```
add_library(huffman_encoder HCTree.cpp)

target_include_directories(huffman_encoder PUBLIC .)
```

Note: difference between PUBLIC and PRIVATE is that with PUBLIC, the build targets that use the huffman_encoder library can access the header files in this include directory, with PRIVATE they cannot access them.

CMake: add_executable()

This defines a set of source files that will be compiled into an executable. You can list dependencies.

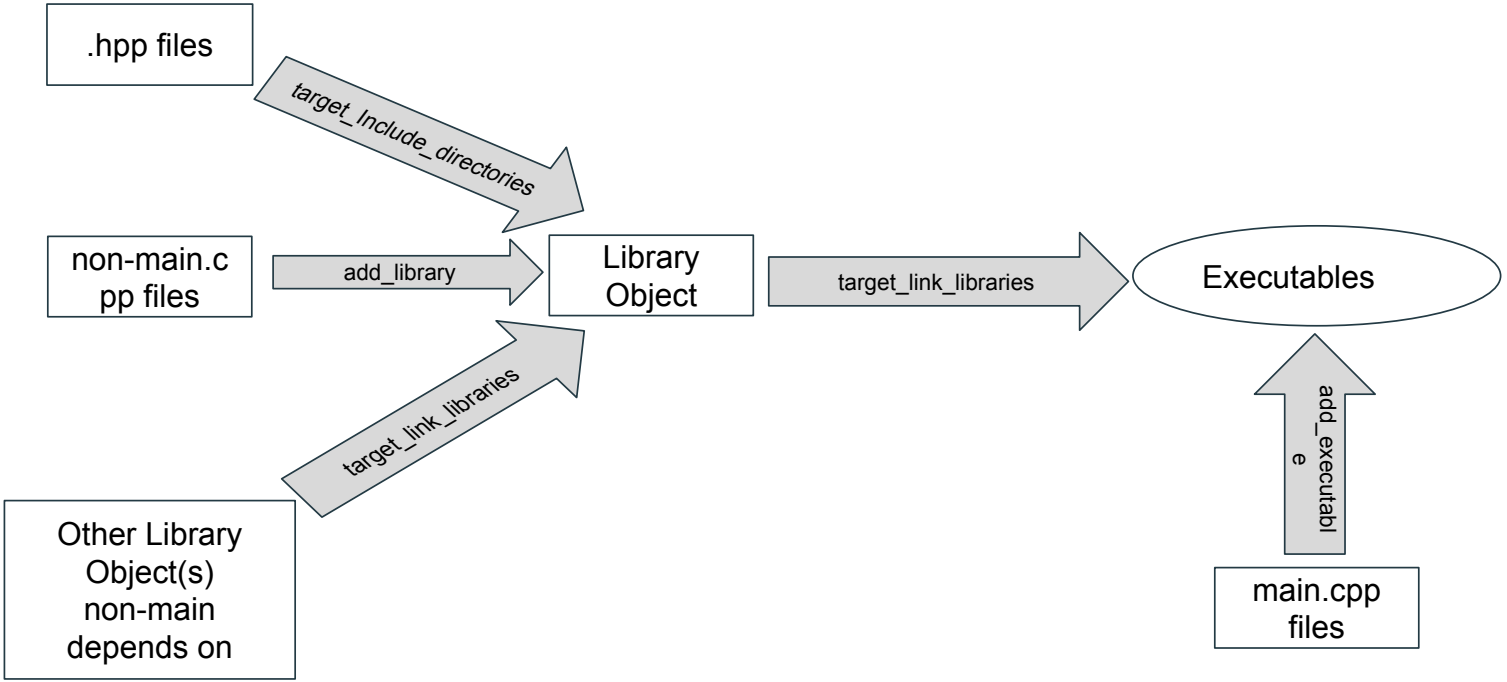
```
build
└── src
    └── uncompress
```

CMake:

```
add_executable(uncompress uncompress.cpp)

target_link_libraries(uncompress PRIVATE huffman_encoder)
```

TO SUM UP: **CMAKE**



C++ File I/O Operation

C++ File I/O operations

class	default mode parameter
ofstream: Stream class to write to files	ios::out
ifstream: Stream class to read from files	ios::in
fstream: Stream class to both read and write from/to files	ios::in | ios::out

Syntax: open (filename, mode);

Modes	Definition
ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content to the current content of the file.
ios::trunc	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

You can combine two or more of these values by ORing them together.

f stands for “file”, there is also a i/ostringstream which you’ll be using in your unit tests. And there is the more generic i/ostream which can accept both the file stream and the stringstream types.

Read the Code

```
int main () {
    ofstream myfile;
    myfile.open("example.txt");
    myfile << "Hello World.\n";
    myfile.close();
    return 0;
}
```

a) Print Hello World on the console screen

b) Will search for existing file example.txt and only if that file exists it will write Hello World to it. Else it will ignore

c) If the file exists then it will write Hello World to it. Else will throw an error

d) If the file exists then it will write Hello World to it. Else will create a new file by the name example.txt and write the content to it.

e) Error, the mode is not specified

Read the Code

```
int main () {
    ofstream myfile;
    myfile.open("example.txt");
    myfile << "Hello World.\n";
    myfile.close();
    return 0;
}
```

a) Print Hello World on the console screen

b) Will search for existing file example.txt and only if that file exists it will write Hello World to it. Else it will ignore

c) If the file exists then it will write Hello World to it. Else will throw an error

d) If the file exists then it will write Hello World to it. Else will create a new file by the name example.txt and write the content to it.

e) Error, the mode is not specified

Happy Coding!

GDB CHEATSHEET

Q: What is GDB?
GDB is the GNU Debugger, a portable system that runs on many Unix-like systems. It is especially useful when debugging C/C++ programs, such as in CSE 100!

Starting GDB:
GDB takes in the name of your program's executable:

```
gdb ./QueueTest
```

Breakpoints in GDB:

- Breakpoints are one of GDB's most useful functionality!
- Breakpoints allow you to stop the execution of your program so you can go through each line step by step.
- The common practice is to set breakpoints **before** running your program. This allows your program to stop at a certain line when running it

```
(b)reak filename:lineNumber
(b)reak filename:functionName
```

Running your program in GDB:
`run` or `r`

Runs your program until a breakpoint or finish.
However, if your executable takes in input or input files, here is an alternative approach:

`r f1 f2 f3` runs your executable with `f1 f2` and `f3` as input files

Running your program in GDB:
`continue` or `c`

Once your program is running, runs your program until the next breakpoint or until finish.
Note: Especially useful if you don't want to "next" or "step" your way through several recursive calls

Next in GDB:
`next` or `n`

Executes the next line of your program, displays the "next" line to be executed after that line

Step in GDB:
`step` or `s`

Steps into a function call. If there is no function call, executes the same functionality as next.

Print in GDB:
`print` or `p`

Prints the current values of variables in your program! Extremely useful debugging feature (: There are many variations of arguments to the command, use a search engine to see more.

```
print argv[i]
print i
print myPtr      → Prints the address myPtr holds
print *myPtr     → Prints the object myPtr is pointing to
```

You can also use one of these format specifiers, which allow you to see your output in non-decimal format if needed.

```
print i          → 2
print/t i        → 10
```

Backtrace in GDB:
`backtrace` or `bt`

Oh no! Your program segfaulted): GDB has a command `backtrace` which prints out the stack trace from your program until failure.

This is especially useful to see which functions/lines your program went through before failure.

Notes:

- `Ctrl-d` or `quit` exits the program
- GDB remembers the last command you executed, so you can continue to press `return` if you're executing the same command over and over.
- Here's a more comprehensive cheatsheet as well!

CSE 100 Project 1 Final Submission
True Compression with Bitwise i/o and Header Design

Final Deadline: **Tuesday, May. 26th, 11:59 pm**

NOTE: For the final submission there will be hidden test cases run after the deadline

Your major tasks in this part include:

1. Write your CMakeLists.txt in the bitstream folder
2. Implement bitwise i/o with a buffer to allow for true compression
3. Test bitstream using the completed executable bitconverter
4. Implement encode and decode with bitwise i/o in HCTree to perform true compression
5. Design an efficient header so that it takes as little space as possible in the compressed file

1. Write CMakeLists.txt

For final submission, you need to write your own CMakeLists.txt files for **all the subdirectories** in src/bitStream. You also need to uncomment the lines that were commented out in all other CMakeLists.txts to connect to your subdirectory part of CMakeLists.

We have listed a summary of the most important concepts in this doc. The full CMake documentation can be found here.

2. Implementing BitOutputStream and BitInputStream

The smallest i/o unit in C++ is a byte by default, so to allow for reading and writing data bit by bit, we need to implement our own bitwise i/o program. The general idea of achieving bitwise i/o is to first use a buffer to store the sequence of bytes being read from an istream or written to an ostream. Then perform the proper bitwise operations to read/write each bit in the buffer. In this part, you will implement a bit i/o stream that uses a buffer with the size given by the user.

the size of buffer: User given

The purpose of using an arbitrary size buffer to perform bit i/o is to potentially speed up the program by using larger buffers. For programs involving heavy i/o tasks, since i/o is generally an expensive operation, outputting data one byte at a time can be very costly. Instead, by using a larger size buffer and read/write the entire chunk of data in the buffer at once, you can easily gain some performance improvement.

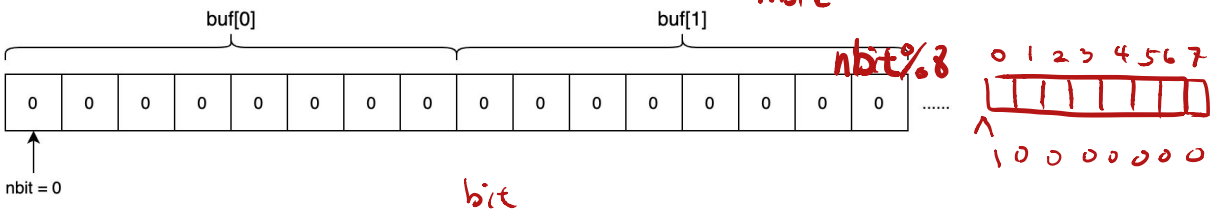
BitOutputStream.hpp, BitOutputStream.cpp

You will implement the following methods based on the idea below (descriptions of methods are in the the files):

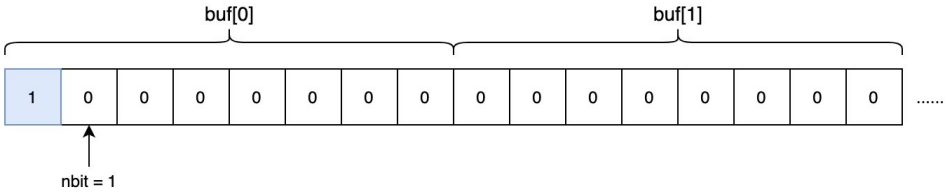
BitOutputStream.hpp, BitOutputStream.cpp

- `BitOutputStream(ostream& os, unsigned int bufSize)`
- `void flush()`
- `void writeBit(unsigned int i)`
- `~BitOutputStream()`

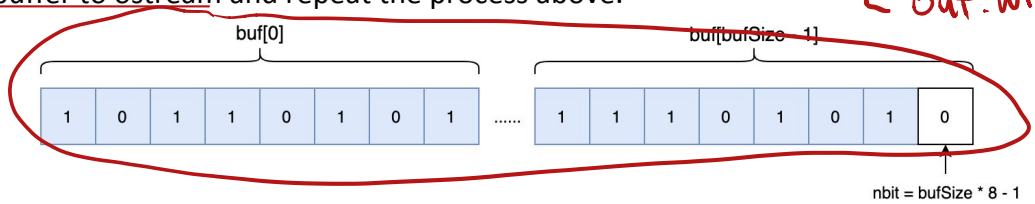
The buffer in BitOutputStream should be a char array with the size defined by the user. The char array should be zero-filled at first, and "nbit" is the number of bits written so far, which can be used as an index that points to the next bit location to write.



After calling writeBit(1), the first bit in the first byte is now 1, and nbit = 1;

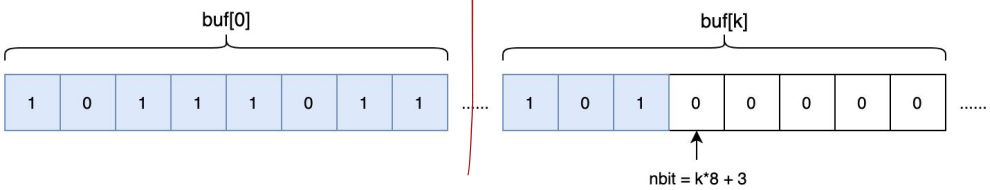


After calling writeBit() for bufSize * 8 - 1 times, nbit = bufSize * 8 - 1. The entire buffer will be full **after the next call to writeBit (when nbit = bufSize * 8)**, then the program should write the entire buffer to ostream and repeat the process above.



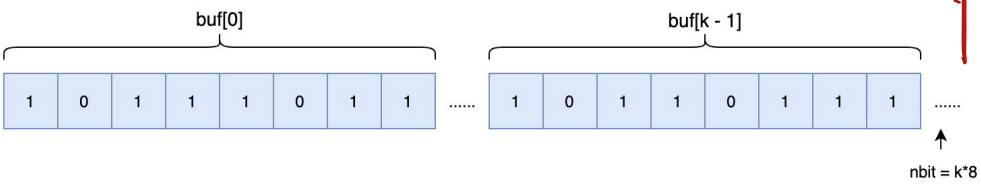
Once the user is done writing bits, the user will call flush() to flush any remaining bits in the buffer to the ostream. Based on the value of nbit, the program needs to determine the number of bytes to write to the ostream, there are two cases:

1. The "current byte" was partially filled. For example, if the first k bytes were filled (from index 0 to index k - 1), but the k + 1 byte (at index k) was not entirely filled.



Then the program should flush k + 1 bytes in total (from index 0 to index k), with the last byte (at index k) containing padding 0s.

2. ~~The "current byte" was entirely filled.~~ For example, if the first k bytes were filled (from index 0 to index k - 1) and the k + 1 byte (at index k) was not filled at all.



Then the program should flush k bytes in total (from index 0 to index k - 1)

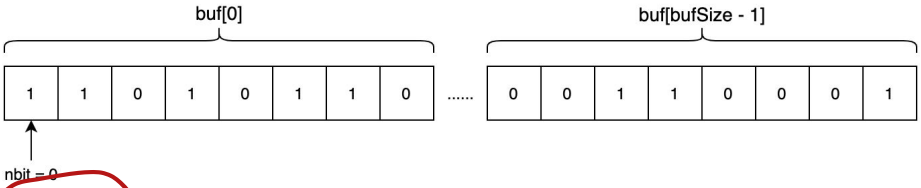
BitInputStream.hpp, BitInputStream.cpp

You will implement the following methods based on the idea below (descriptions of methods are in the the files):

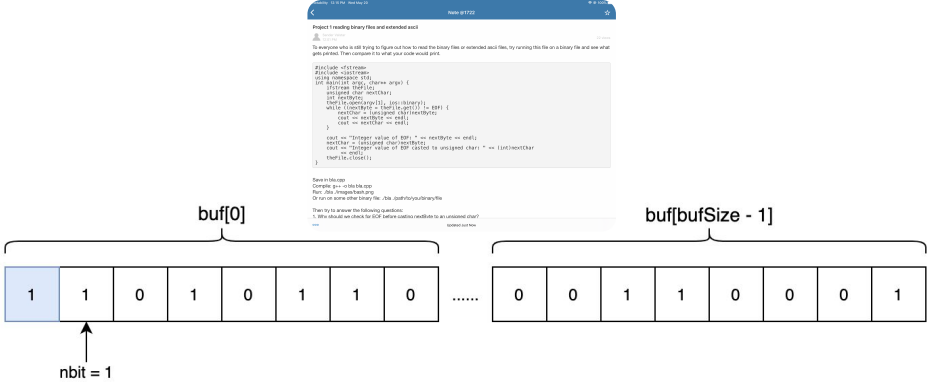
`BitInputStream.hpp, BitInputStream.cpp`

- `BitInputStream(istream& is, unsigned int bufSize)`
- `void fill()`
- `bool atEndOfFile()`
- `unsigned int readBit()`
- `~BitInputStream()`

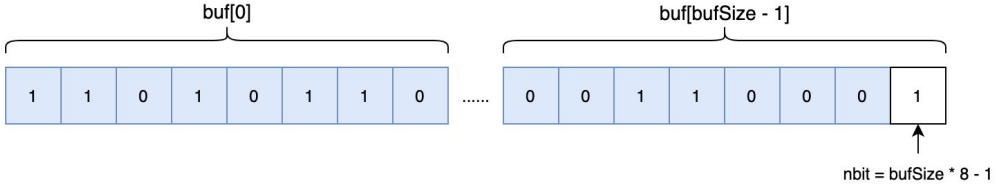
The buffer in BitInputStream is also a char array with a size defined by the user. "nbit" is the number of bits read so far, which can be used as an index that points to the next bit location to read. After initializing the char array with the proper size, the program should first read from the istream to fill the buffer. (assume there are at least bufSize number of bytes in the istream for now)



Now calling readBit() should return 1, and nbit is increased by 1:



After calling readBit() for bufSize * 8 - 1 times, nbit = bufSize * 8 - 1. The entire buffer will be read **after the next call to readBit (when nbit = bufSize * 8)**, then refill the buffer from istream and repeat the process.



Once the istream tried to read more bytes than the remaining bytes in the file and the buffer was also fully read, then the helper atEndOfFile() in BitInputStream should return true.

Let the number of remaining bytes in the istream be n. If n is less than bufSize, we can still read those n remaining bytes into the first n bytes of the buffer by calling read(). We can then use the value of istream to tell us if istream tried to read more bytes than the remaining bytes in the file, and use gcount() to know the number of bytes successfully read to buffer:

```
in.read(buf, bufSize); // reading the last n bytes, n < bufSize
if (!in) cout << "istream tried to read more than remaining bytes" << end;
int numBytesRead = in.gcount(); // number of bytes successfully read to buffer
```

In either of the following cases, we know the buffer was also fully read:
1. Number of bytes successfully read to the buffer = 0.
2. Otherwise, there were some bytes successfully read to the buffer. When nbit = (number of bytes successfully read to the buffer) * 8, we know the buffer is fully read.

3. Test bitstream using bitconverter

The bitconverter is a completed program that can convert between ascii chars and their binary representations. There are two options (-b: toBit and -c: toChar) to run the program:

```
./build/src/bitconverter -b 100 charFile.txt bitFile.txt
```

With -b option, each ascii char in charFile.txt will be translated to its binary representation in the ascii table. For example, a charFile with "a" will be translated to a bitFile with "01100001".

The number in the second argument specifies the buffer size used in BitInputStream. With the -b option, this program is essentially testing your BitInputStream.

```
./build/src/bitconverter -c 100 bitFile.txt charFile.txt
```

With -c option, each 8 bits in bitFile.txt will be translated to the corresponding symbol in ascii table. For example, a bitFile with "01100001" will be translated to a charFile with "a". You may assume the bitFile only contains chars either '0's or '1's

If the number of bits in bitFile.txt is not divisible by 8, then the last char in the charFile can contain some padding 0s. For example, a bitFile with "01101" will be translated to a charFile with "h" which has binary representation 01101000 (the last 3 bits are padding 0s).

The number in the second argument specifies the buffer size used in BitInputStream. With -c option, this program is essentially testing your BitOutputStream.

For any arbitrary text file as the charFile.txt above and any nonnegative buffer size, your program should produce same output as the reference solution:

```
./build/src/bitconverter -b 100 data/bitstream_1.txt myBitFile.txt
./solution-bitconverter.executable -b 100 data/bitstream_1.txt refBitFile.txt
diff myBitFile.txt refBitFile.txt
```

Your converter should also be able to get the original file back:

```
./build/src/bitconverter -b 100 charFile.txt bitFile.txt
./build/src/bitconverter -c 100 bitFile.txt charBack.txt
diff charFile.txt charBack.txt
```

It is important that you first pass this test before implementing true compression, as the compression program heavily depends on working bitstreams. The grading cases used in autograder are listed in testing cases.

4. Implementing Encode and Decode with Bitwise i/o

Now that you have implemented bitwise i/o, proceed to implement the following functions in HCTree to allow encode and decode using bits (you can uncomment them in hpp and cpp files):

- `void encode(byte symbol, ostream& out) const`
- `byte decode(istream& in) const`

Implement trueCompression and trueDecompress

Now implement the trueCompression() in compress.cpp and trueDecompression() in uncompress.cpp by using bitwise i/o functions in your HCTree. For final submission, you should choose the buffer size to be 4000 for both the input and output streams.

```
./build/src/compress data/toCompress.txt compressed.txt
./build/src/uncompress compressed.txt decompressed.txt
```

When running the above commands (without the --ascii flag), your program should perform true compression/decompression. The compressed file should now contain real bits instead of '0' and '1' ascii chars. (You may use the flag "isAsciiOutput" to determine whether to call pseudo or true compression/decompression.)

Once you finish implementing this part, make sure your decompressed file is the same as the original file. With the correct implementation, given input files such as file_5.txt (the novel *War and Peace*), your compression program should now be able to produce a compressed file that is smaller than the original file.

The reference solution contains an option --freqsHeader (make sure you **don't** add this option to your submission, it is only for grading purposes) that makes the program output the naive header in the checkpoint with actual encoding bits.

```
./solution-compress.executable --freqsHeader data/toCompress.txt compressed.txt
./solution-uncompress.executable --freqsHeader compressed.txt decompressed.txt
```

With option --freqsHeader, the size of the compressed file from the reference solution should be the baseline in your final submission. This means for the same original file, your compressed file should be no larger than the compressed file from the reference solution running with --freqsHeader.

You can refer to testing cases for detailed requirements and grading cases.

5. Efficient Header Design

We can see that the header used in the checkpoint solution might take some space: before the program even starts to encode any symbol, the header had already taken at least 500 bytes of space (It could take more space when certain frequencies are large). And for files of size around 500 bytes, this is especially expensive because you might end up with a larger compressed file.

Your last task in this assignment is to further compress the input file by implementing a space efficient header to replace the header used in checkpoint. In order to earn full credit for the header design, your compression and decompression programs must work correctly in the first place, then **the header of the compressed file produced by your program must be smaller than the header produced by the reference solution for most of the files when running without any options**. For a file with really small size where each symbol has low frequency, your program is not required to beat the reference solution (however, your header size should be really close to the reference solution with only a few bytes off).

In particular, in order to get any credits for header design, your program should first pass the File_5 correctness test. Then for full credit, the header size of your compressed file from file_5.txt should be smaller than the one produced by the reference solution when running without any options.

Partial credit will be given if your solution's header is bigger than the reference solution's header by no more than 10% (within 110%). The cutoff is 10%, which means no credit for header design will be given if the header size produced by your solution is more than 10% larger than the reference solution.

When running the reference solution, it will print out the header size so you may use that information to compare with your header size. To check the size of the compressed file, you can simply run "`ls -l filename`". ls -l filename

Note that because people have different ways to implement the header, the compressed files are just some mysterious bits for others, so we have to make a reasonable assumption when determining the size of your header: **given the same input file, the length of the encoding bits of your solution and the reference solution should always be the same**, because Huffman Coding gives the optimal encoding length. Therefore, when grading your header size for the same input file, we simply use the following formula based on the assumption above:

```
size of your header = size of your compressed file - size of the non-header part of reference compressed file
```

Before you get started to implement the efficient header, remember to save your work now by making a git commit. Also make sure you have already tested your program comprehensively.

Reference solution header that you need to beat

Since you are required to beat the reference solution, it might be helpful to first know how it implements the header, so that you won't do the same thing in your PA. The idea of a reference

solution's header is basically removing all redundant information. It does the following to produce the header of the compressed file:

1. Find the maximum frequency of all symbols. Based on this number, determine the number of bits needed to store each frequency (For example, if the maximum frequency is 14, then you only need 4 bits to represent any frequency in the original file, since 4 bits can store numbers up to 2^4 - 1 = 15). Then let this number be `numBits`.
2. Find the number of non-zero frequency symbols.
3. As one particular symbol can occur up to 2 billion times as stated, the maximum `numBits` is about 31, which requires 5 bits to store. Then we use fixed 5 bits to store `numBits` in the beginning of the header.
4. The maximum number of non-zero frequency symbols is 256, which requires 8 bits to store. Then we use fixed 8 bits to store it in the header.
5. In the remaining part of the header, for each non-zero frequency symbol, output 8 bits to represent the symbol, followed by `numBits` bits to represent its frequency.

For example, given a file with content "aaaaaabbbcccc", the header should look like the following in bits with each char like 'a' representing 8 bits:

00011 00000011 a110 b011 c100

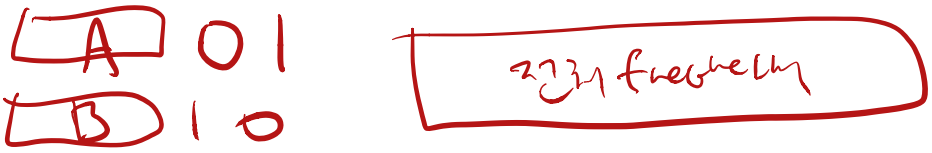
After some calculations, it is easy to see that the header above takes 46 bits to store.

In the decompress program, this would be interpreted as: `numBits` = 3, number of non-zero frequency char = 3. So we should read 3 blocks of char-freq: in each block, first read 8 bits to get the symbol, then read `numBits` bits to get its frequency.

Hint:

All the headers we have seen are based on the assumption that we need to represent the frequency vector such that the decompression program can use it to rebuild the tree. Using this idea, the reference solution is probably the best we can do.

That means if you want to beat the reference solution, you should think about this problem from another perspective: instead of focusing on the frequency vector, if somehow we know the correct structure of the original HCTree and are able to reconstruct it, then decode and encode should also work even without knowing the frequency of each symbol, as the frequency of each symbol is only used in building HCTree.



So that boils down to the following problem: Can we encode the structure of the HCTree in a sequence of bits? And based on this sequence of bits, can we reconstruct the original HCTree?

Testing Cases and Grading Breakdown

The testing cases and score distribution below should provide guidance on your own testing. **We've provided some test files that we are using in the autograder for you, make sure you read "Diffing output" to understand the requirements.** Those test cases will be set as visible when you submit your PA on gradescope. You can find the provided test files in the data folder.

Note that your compression program should work regardless of the size and type of the files. You can only assume the input file is less than 4GB, which means a particular symbol may appear 4 billion times.

By the time you submit Project1 final, please complete the following survey which is worth 5 points towards your Project1 final grade
Week 8 reflection survey

: provided test files in data folder (file with name file_1.txt etc.)
: makeup credit (if you received less than the maximum #points, these points will apply)

Final - 70 pts
In final, to get full credits for each testing file ("File 1" etc):
1. The uncompressed file from your program should match the original file
2. The compressed file should have a size no larger than the compressed file from the reference solution with --freqHeader option enabled.

Testing Cases	Description	Points
File 1	An empty file with no content	2
File 2	A file contains only one repeating single kind of chars (all newlines)	3
File 3	A file with alphabet letters	5
File 4	A file with symbols possibly from extended ASCII table, with ASCII value larger than 127.	5
File 5	The novel *War and Peace*	5

File 6	Non-text file such as binary file (executable), image, and video	5
File 7	Many of one symbol, and a little of all other symbols	5
File 8	A large file with size about 60 MB	5
BitStream 1	Run provided bitconverter with -b can convert a sequence of chars to a corresponding sequence of binary representation. (The non-negative buffer size can be less than, equal to, or greater than the size of input file)	5
BitStream 2	Run provided bitconverter with -c can convert a sequence of bits to a sequence of chars. (The non-negative buffer size can be less than, equal to, or greater than the size of input file)	5
Header Design	Full credits: the header size of the compressed file from file_5.txt is smaller than the one from the reference solution. Partial credits: the header size is within 110% of the header size from the reference solution.	15
Valgrind	No memory leaks in compress.cpp and uncompress.cpp	3
Code formatting	Passing the autoformatting check	1
Student tests	Passing your own tests	1
Reflection survey	Points for completing the reflection survey	5
Code coverage	90% of line coverage for encoder and bitStream	10