PRIORITY QUEUES
REVIEW OF DATA STRUCTURES
FINAL WRAP UP

Final exam

- Check out the final resource page: https://ucsb-cs24-w18.github.io/exam/e02/
- Diba's office hours and extra hours:
  - Thursday : 11am to 1pm
  - Friday: noon to 1pm
  - Monday: 10 to noon

Goals of this class

- Object oriented programming
- Data structures and operations that they support
  - Arrays
  - Dynamic Arrays
  - Linked lists (single and doubly linked)
  - Stacks
  - Queues
  - Binary Search Trees
  - Binary Coded Trees (tries)
  - Heaps (also known as priority queue)
- Ability to select the right data structure for your problem by knowing:
  - Operations supported by the data structure
  - Big-O running time of these operations (not from memory but through analysis)
- Ability to implement each of these data structures in C++
- Ability to use the C++ STL implementations of these data structures in your algorithms.

Priority Queues in C++

A C++ **`priority_queue`** is a generic container, and can hold any kind of thing as specified with a template parameter when it is created: for example **`HCNode`**s, or pointers to **`HCNode`**s, etc.

```
#include <queue>
std::priority_queue<HCNode> p;
```

- You can extract object of highest priority in O(log N)
- To determine priority: objects in a priority queue must be comparable to each other
- By default, a **`priority_queue<T>`** uses **`operator<`** defined for objects of type T:
  - **if `a < b`, `b` is taken to have higher priority than `a`**

std::priority_queue methods

A C++ `priority_queue` is a generic container, and can hold any kind of thing as specified with a template parameter when it is created: for example `ints`, `Nodes`, or pointers to `Nodes`, etc.

```
priority_queue<int> pq;
```

Methods:

```
* push()   //insert
* pop()    //delete max priority item
* top()    //get max priority item
```

- You can extract object of highest priority in O(log N)
- To determine priority: objects in a priority queue must be comparable to each other

std::priority_queue template arguments

The template for priority_queue takes 3 arguments:

```
template <
      class T,
      class Container= vector<T>,
      class Compare = less <typname Container::value_type>
      > class priority_queue;
```

- The first is the type of the elements contained in the queue.
- If it is the only template argument used, the remaining 2 get their default values:
  - a **vector<T>** is used as the internal store for the queue,
  - **less is a comparator** class that provides priority comparisons

Comparator class

- The documentation says of the third template argument:
- Compare: Comparison class: A class such that the expression comp(a,b), where comp is an object of this class and a and b are elements of the container, returns true if a is to be placed earlier than b in a ordering operation. This can be a class implementing a function call operator.

- **The default std::less is a comparator** class that provides priority comparisons

```
less<int> ls;
```

if `ls(a,b)` is `true`, **a has less priority over b**

```
class less{
        bool operator()(T& a, T & b) const {
  return a<b;
 }
};
```

Comparator class

Here's how to define a class implementing the function call operator operator() that performs the required comparison:

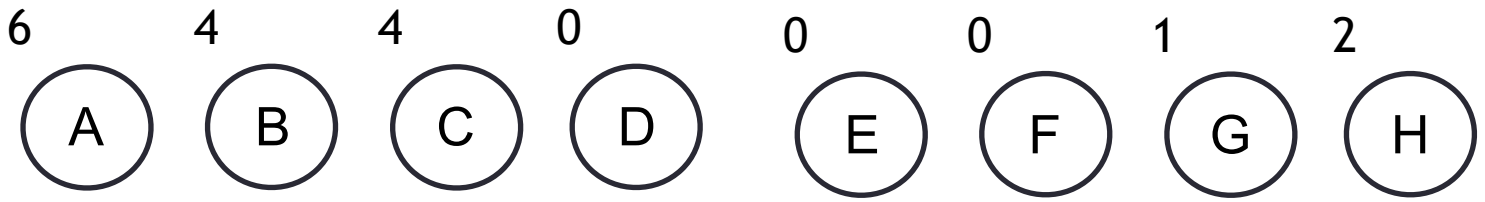
```
class NodePtrCompare {
  bool operator()(Node* & lhs, Node* & rhs) const {
    // dereference the pointers and use operator<
    return *lhs < *rhs;
  }
};
```

Application of priority_queues (heaps)

Demo: Compare selection sort and heap sort algorithms

Huffman's algorithm: Bottom up construction

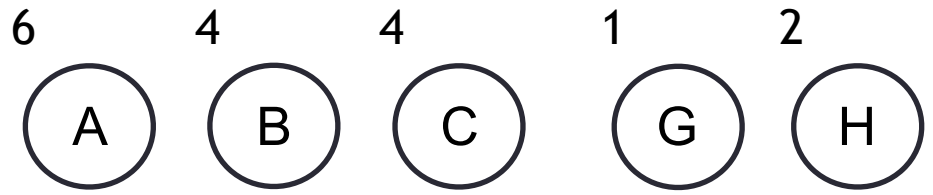
- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node



A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2

Huffman's algorithm: Bottom up construction

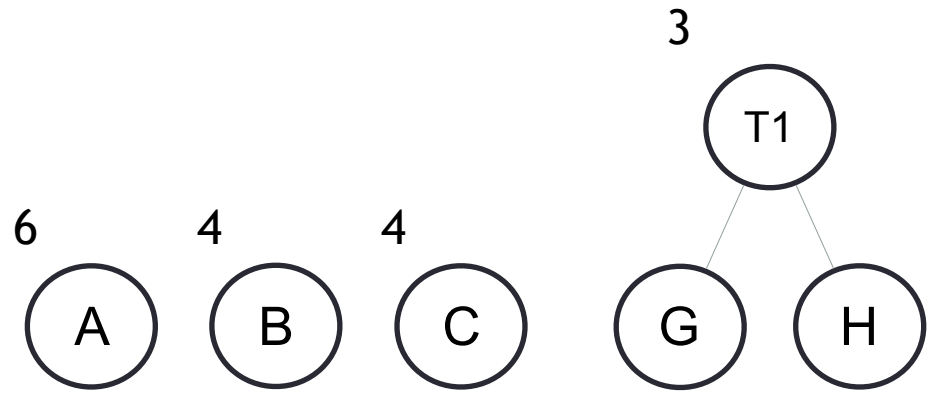
- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them



A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2

Huffman's algorithm: Bottom up construction

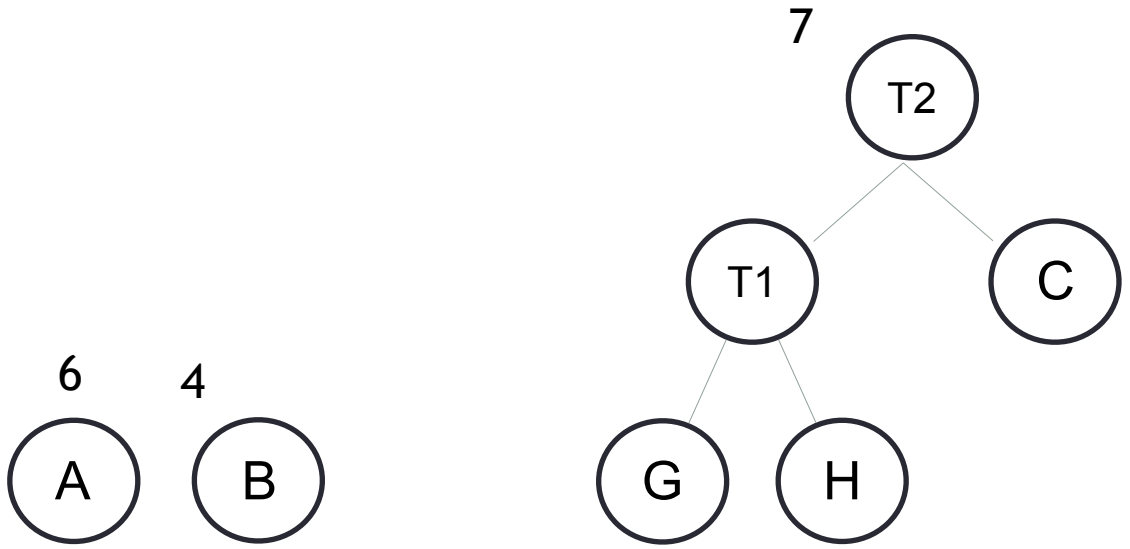
- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them



A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2

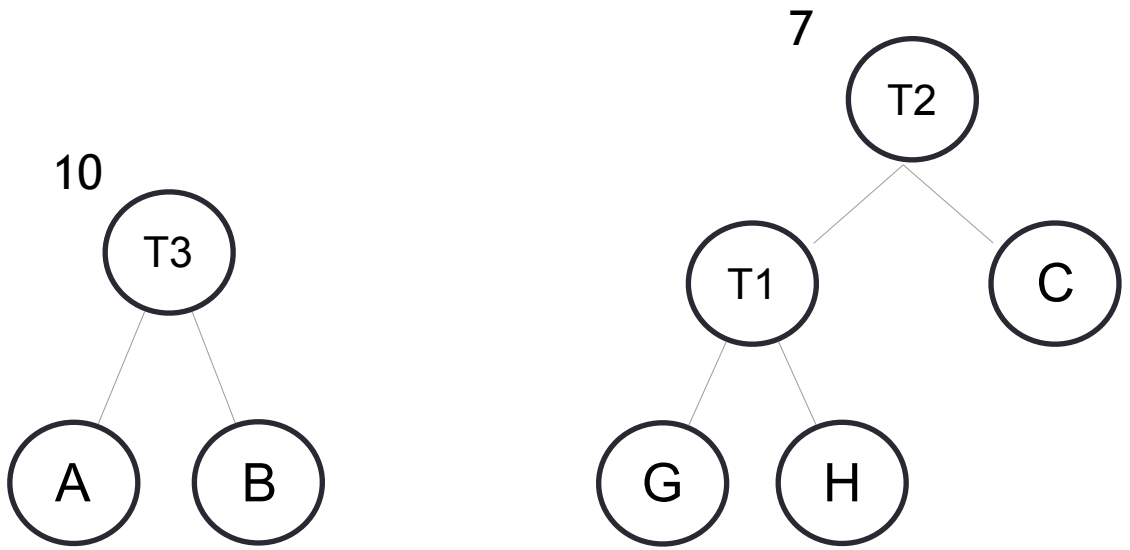
Huffman's algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them
- Repeat until all nodes are in the tree



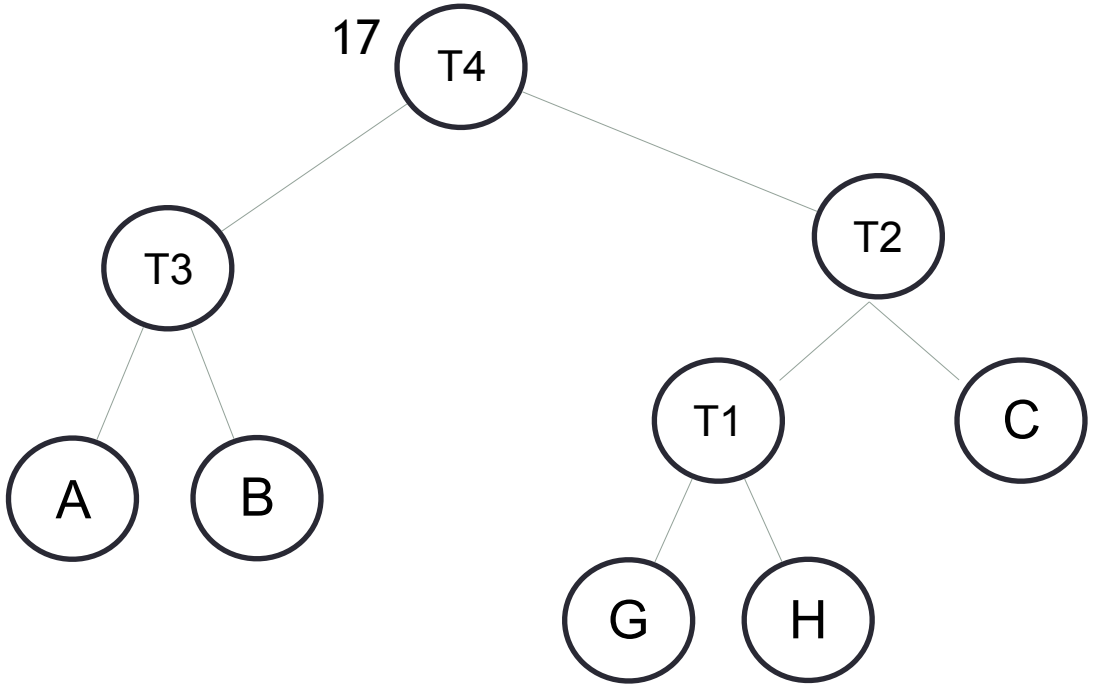
Huffman's algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them
- Repeat until all nodes are in the tree

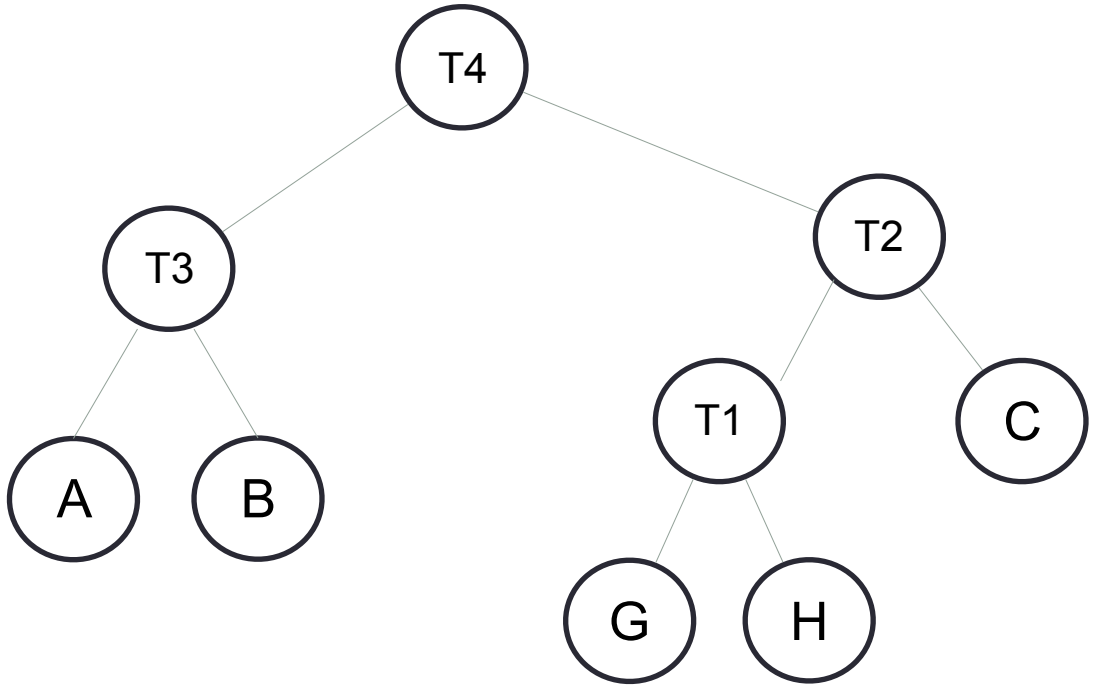


Huffman's algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them
- Repeat until all nodes are in the tree



Encoding a symbol- think implementation!



- Compression using trees:
  - Devise a “good” code/tree
  - Encode symbols using this tree

A very bad way is to start at the root and search down the tree until you find the symbol you are trying to encode, why?

The suitability of Priority Queues for our problem

- In the Huffman problem we are doing repeated inserts and delete-min!
- Perfect setting to use a Heap data structure.
- The C++ STL container class: priority_queue has a Heap implementation.
- Priority Queue and Heap are synonymous because PQs are usually implemented with Heaps

Use a priority queue (heap) to hold the forest of trees in step 2.
How long does it take to find the min element in a heap?

A. O(1)
B. O(log(n))
C. O(n)

Building the tree: Huffman's algorithm

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.

You know how to create a tree. But how do you maintain the forest? Choose the best data structure/ADT:

A. A list

B. A BST

C. A priority queue. Repeated inserts, and delete mins So, a heap is better.

A balanced BST will work too.

Data structure Comparison

	Insert	Search	Min	Max	Delete min	Delete max	Delete (any)
Sorted array	O(N)	O(logN)	O(1)	O(1)	O(N) if ascending order, else O(1)	O(1) if ascending, else O(N)	O(logN) to find, O(N) to delete
Unsorted array	O(1)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
Sorted linked list (assume access to both head and tail)	O(N)	O(N)	O(1)	O(1)	O(1)	O(1)	O(N) to find, O(1) to delete
Unsorted linked list	O(1)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N) to find, O(1) to delete
Stack	O(1) - only insert to top	Not supported	Not supported	Not supported	Not supported	Not supported	O(1) - Only the element on top of the stack
Queue	O(1) - only to the rear of the queue	Not supported	Not supported	Not supported	Not supported	Not supported	O(1) - only the element at the front of the queue
BST (unbalanced)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
BST (balanced)	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)	O(logN)
Min Heap	O(logN)	Not supported	O(1)	Not supported	O(logN)	Not supported	O(logN)
Max Heap	O(logN)	Not supported	Not supported	O(1)	Not supported	O(logN)	O(logN)

Data structure Comparison

Which of the following pairs of data structures, which of the pair is the better choice for.

- Inserting a list of *sorted* elements, elements are inserted in the order they appear in the list (worst case):

A. Linked list
B. Simple binary search tree
C. They are about equal

Answer is A : linked list is better
Linked list is better because we can always insert at the head of the list: O(1)
BST is worse because the nodes are sorted. Whether the elements appear in ascending or descending order, the worst case height of the BST will be O(N) and inserts will be O(N)

- In-order traversal of elements:

A. Heap-> We cannot traverse the elements of a heap in sorted order unless they are deleted from the heap (which is not the same as traversal)
B. Balanced binary search tree
C. They are about equal

Answer is B

Data structure Comparison

Which of the following data structures is the better choice for.

- (Big-O) time to find an element:

A. Sorted array
B. Balanced BST
C. They are about equal

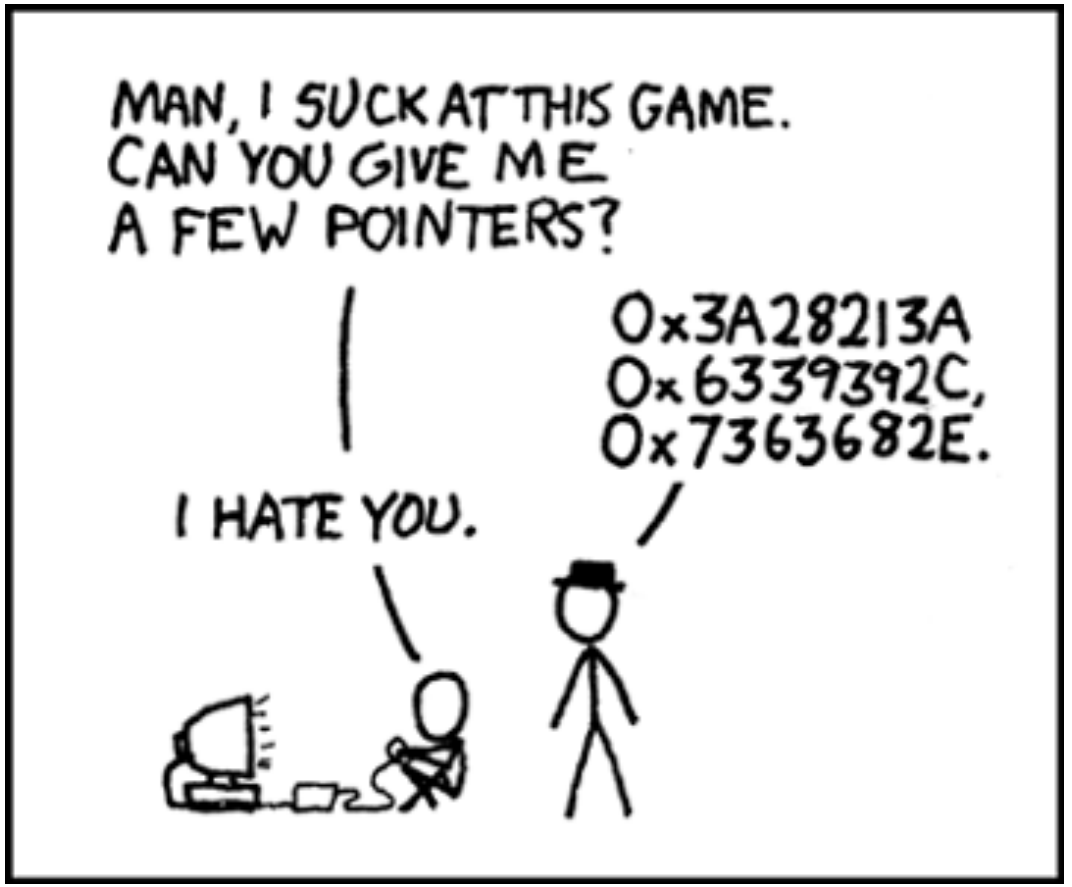
Answer is C -> O(log N)

- Requires less space:

A. Heap
B. BST
C. They are about equal

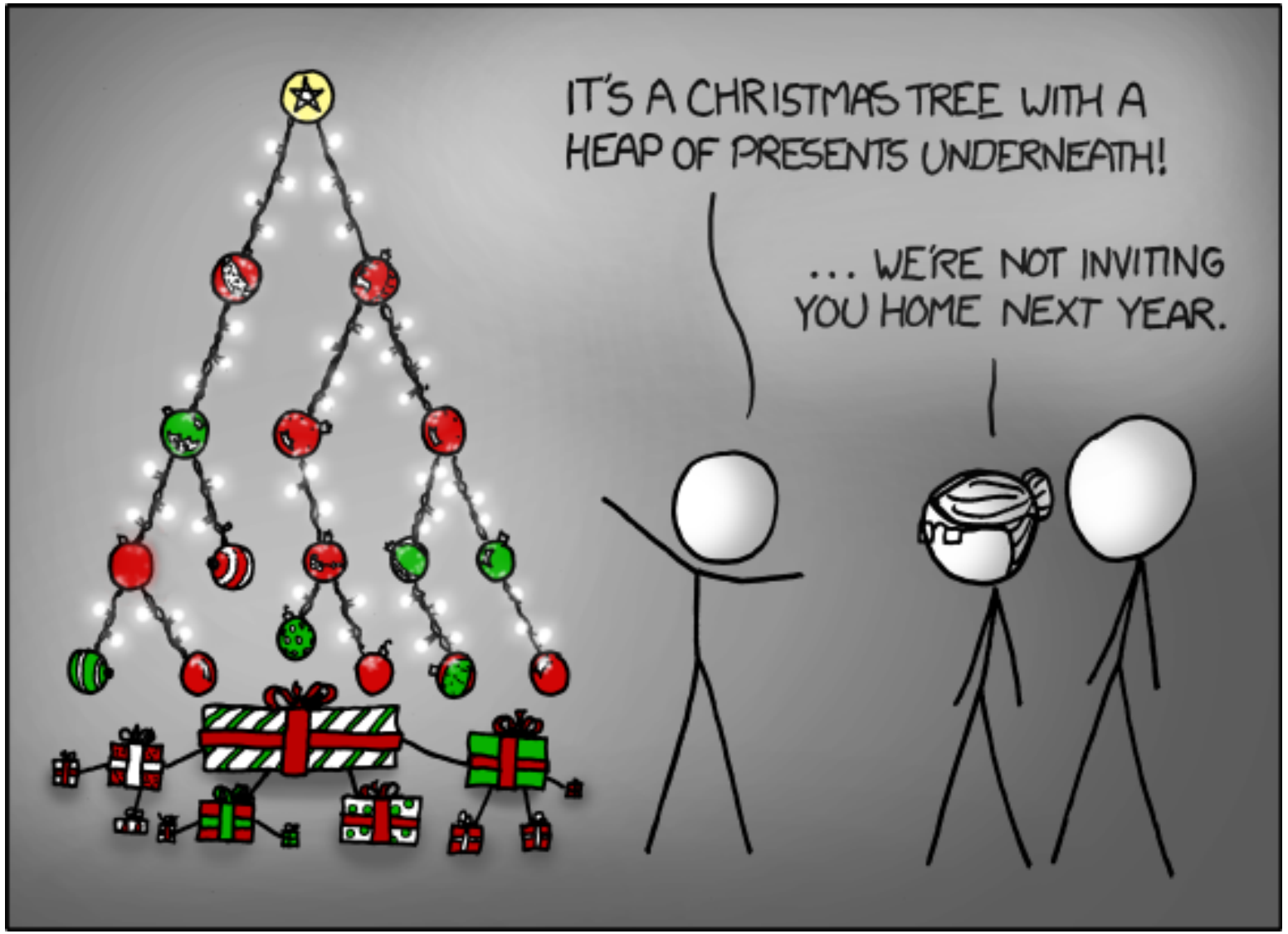
Answer is A -> Heap can be implemented as an array which is more space efficient

Some comic relief



HTTP://XKCD.COM/138/

Some comic relief…



http://xkcd.com/835/

Good luck with the final!