Lecture 9

- Time and space cost analysis of Huffman coding
- Huffman coding tree implementation issues
- Priority queues and priority queue implementations
- Heaps
- Dynamic data and array representations for Huffman trees

Reading: Weiss Ch. 6, Ch. 10.1.2

Huffman code trees

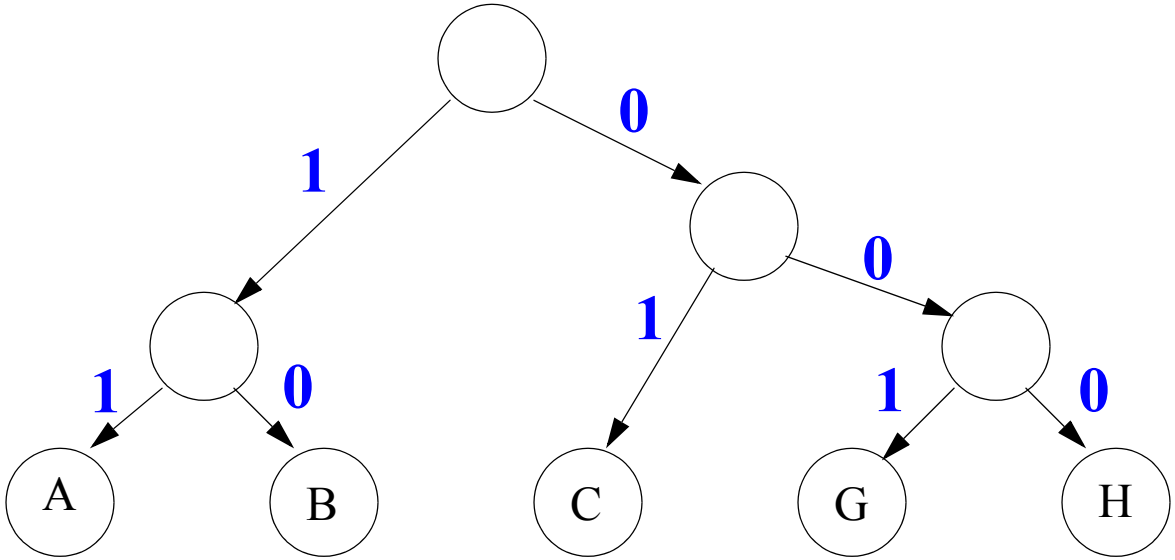
- Last time, we discussed the Huffman coding algorithm
- The Huffman algorithm constructs a tree (a binary trie) which represents a code
- This tree (or equivalent information) is used to code items from the input; the output is a sequence of bits
- An identical tree (or equivalent information) must be used to decode that sequence of bits, to get back the original input that was coded!
- We will look at some issues in implementing a Huffman code tree

Basic properties of a Huffman code tree

- Suppose there are N different symbols in the input sequence (some of them may appear more than once)
  - For example, the input sequence
    THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
    has length 43, and contains 27 different symbols: The capital letters A-Z, and space
- The Huffman trie-construction algorithm starts with a forest of N one-node trees, and then, while there is more than one tree in the forest:
  - removes two trees from the forest,
  - makes them the left and right subtrees of a new parent node, and
  - returns the new tree to the forest
- The result is a "full" binary tree: every non-leaf node has exactly 2 children (why?!)
- Questions: in the resulting tree as functions of N:
  - How many leaf nodes are there? _____
  - How many internal (non-leaf) nodes are there? _____
  - So, how many nodes in the code tree? _____

Coding with a Huffman tree

- Suppose you have a Huffman coding tree like this:



- ... and you want to code the letter A

- The code for letter A is the sequence of bits labeling edges on the path from the root of the tree to the leaf containing A

- How to find that path?

Coding with a Huffman tree, cont'd

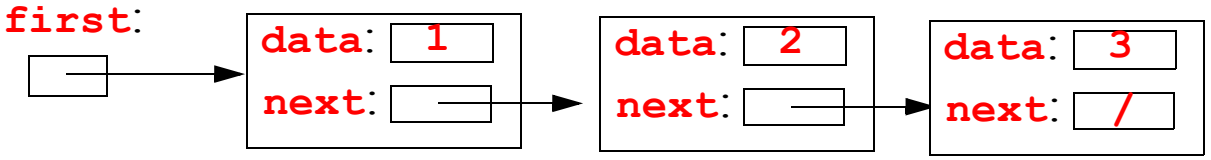
- A not very good way to do it is to start at the root, and try to find a path to the leaf containing A
- (You *could* do that, but worst case you would have to traverse the entire tree before finally finding the leaf you want, taking $\Omega(N)$ steps)
- A better way: start at the leaf containing A, follow the (unique, since in a tree every non-root node has a unique parent) path to the root of the tree keeping track of the bits on the path... and then reverse that sequence of bits
- A clever way to do that uses recursion
- To see how to do that, we will take a short detour into linked lists...

Traversing a data structure

- A data structure contains elements, which contain data
- *Traversing* or *iterating over* a data structure means: "visiting" or "touching" the elements of the structure, and doing something with the data
- For example you could have a singly-linked list, with elements that are instances of this class:

```
class LNode {
   int data;
   LNode* next;
}
```

- Suppose `first` is a pointer that points to the first element of a list of `LNode` objects; the last element in the list has a null `next` field



- Traversing that list, from first to last, and printing out data in the elements, would print:
  `1 2 3`

Traversing a linked list

- Suppose `first` is a pointer that points to the first element of a long list of `LNode` objects; the last element in the list has a null `next` field

- To traverse it from first element to last, you could do it iteratively:

```
void traverse(LNode* n) {
   while(n) {
      std::cout << n->data << std::endl;
      n = n->next;
   }
}
```

- Now `traverse(first)` will will traverse the list pointed to by `first`, printing out the data fields, first element to last

- (It is also possible to do this recursively)

Traversing a linked list, in reverse

- Again, suppose `first` is a pointer that points to the first element of a long list of `LNode` objects; the last element in the list has a null `next` field

- But suppose now that you want to traverse it from last element to first
- Here, it is possible to do it iteratively, but it is much neater to do it recursively:

```
// PRE:  n points to a node of a list with null next field
// in its last node; or n is null
// POST: data fields in the list from node n through the last
// node have been printed, in reverse order
void traverse(LNode* n) {
   if(!n) return; // base case: empty list
   traverse(n->next); // traverse the rest of the list, reversed
   std::cout << n->data << std::endl; // print out this elt
}
```

- Now `traverse(first)` will traverse the entire list, last element to first

- (What happens if the 2nd and 3rd statements in the method are exchanged?)

Root-to-leaf path traversals in a Huffman code tree

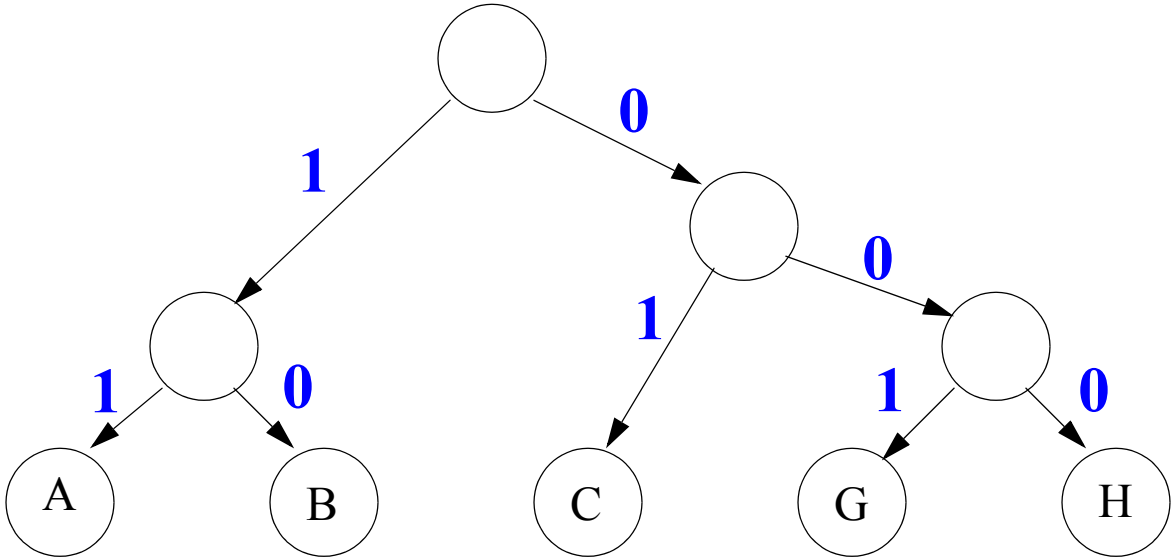
- Consider the nodes from a leaf to the root as a linked list
  - (this requires that each node have a link to its parent: a “parent pointer”)
  - thus, the whole tree can be thought of as a collection of linked lists, which share nodes and links

- Consider the leaf as the first element in the list
  - to find quickly the leaf corresponding to an item to be coded, maintain a table data structure that associates each item to be coded with a pointer to its leaf node in the tree

- Traverse this list, in reverse order. When a node is visited, output “0” or “1” depending on whether it is a “0” or “1” child of its parent
  - this requires that a node somehow specify whether it is a “0” child or a “1” child of its parent

Decoding with a Huffman tree

- Suppose you have a Huffman coding tree like this:



- ... and you want to decode the bit sequence 001

- The letter for that code is contained in the leaf you find at the end of the path starting at the root, following links to “0” or “1” children as given by the bit sequence

- How to find that path?

Decoding with a Huffman tree, cont'd

- Decoding with a Huffman tree is a bit more straightforward than coding
- Start at the root of the tree, and follow links to "0" or "1" children, depending on the next bit in the code
- When you reach a leaf, the symbol you've just decoded is found in it

Decoding path traversals in a Huffman code tree

- Exactly the same tree (or equivalent information) needs to be used for decoding and coding, to ensure that you get out of the code what you put into it

- Do decode, you follow a path from the root of the tree to a leaf
  - this requires that each internal node have links to its “0” and its “1” child

- You output the value of the coded symbol when you reach a leaf
  - this requires that each leaf node contain or point to the value of the corresponding symbol

Nodes in a Huffman code tree

- So we have found that to be useful for both coding and decoding, a node in a Huffman code tree should have:
  - a pointer to its parent (which will be null, if it is the root of the tree)
  - some way to tell whether this is the “0” or “1” child of its parent
  - a pointer to its “0” child and its “1” child (which will both be null, if it is a leaf)
  - the value of the alphabet symbol (if it is a leaf)

- You also should have a fast way, given a symbol to code, to get to the code tree leaf containing that symbol

- There are various ways to implement these requirements...

Ways to implement a tree

- As you know, in computer science, if there's one way to do something, there's infinitely many ways to do it

- When implementing a tree, for example, you can
  - use dynamic data and pointers
    - flexible, general purpose. Clearly the best choice when you don't know the structure or the size of the tree beforehand
  - use an array
    - fast, compact. Usually the best choice when implementing heaps, or other trees with a very regular structure

- You can use either of these to implement a Huffman code tree. We will look at each, but first we'll do some analysis of the time cost of the Huffman algorithm

Huffman's algorithm

0. Determine the count of each symbol in the input message.

1. Create a forest of single-node trees. Each node in the initial forest contains a symbol from the set of possible symbols, together with the count of that symbol in the message to be coded. Symbols with a count of zero are ignored (consider them to be impossible).

2. Loop while there is more than 1 tree in the forest:

2a. Remove the two trees from the forest that have the lowest count contained in their roots.

2b. Create a new node that will be the root of a new tree. This new tree will have those two trees just removed in step 2a as left and right subtrees. The count in the root of this new tree will be the sum of the counts in the roots of its subtrees. Label the edge from this new root to its left subtree "1", and label the edge to its right subtree "0".

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.

Analysis of Huffman's algorithm

- Suppose the input message is a sequence of length K drawn from an alphabet of N symbols
  - These two parameters define the size of a Huffman coding problem... so we will define space and time costs in terms of them

Step **0**: time cost O(K)

Step **1**: time cost depends on the data structure used to hold the forest

Step **2**: the loop is executed _____ times

Step **2a**: time cost depends on the data structure used to hold the forest

Step **2b**: time cost O(1)

Step **2c**: time cost depends on the data structure used to hold the forest

Step **3**: time cost O(1)

- Overall time cost will depend on the performance of the data structure used to hold the forest, so let's look at that in detail

Priority queues and Huffman's algorithm

- The data structure holding the Huffman forest needs to support these operations:
  - Create an initially empty structure
  - Insert a tree into the structure
  - Delete from the structure, and return, the tree with the smallest count in its root

- These are exactly the typical operations provided by the Priority Queue abstract data type, used in many fundamental algorithms:
  - Create an empty Priority Queue
  - Insert an item into the Priority Queue
  - Delete, and return, the item in the Priority Queue with highest priority

Queues and Priority Queues

- An (ordinary) Queue is an abstract data type with these operations:
  - Create an empty Queue
  - Insert an item into the Queue
  - Delete, and return, an item in the Queue
- Of all the items currently in a Queue, the Delete operation must return which one?.....

  (So, a Queue is often called a _____ structure.)

- A Priority Queue is an abstract data type with these operations:
  - Create an empty Priority Queue
  - Insert an item into the Priority Queue
  - Delete, and return, an item in the Priority Queue
- The difference is: in a Priority Queue, the Delete operation must return the item with the highest *priority*
- So, items stored in a Priority Queue must be comparable to each other with respect to their "priority"

Implementing a priority queue using a linked list

- One common way to implement a queue is with a linked list
  - Maintain pointers to the beginning of the list, and the end of the list
  - Insert items at the end; delete items from the beginning (or vice versa)

- A linked list could also be used to implement a priority queue
  - Maintain the list in sorted order, with the "highest priority" item first

- Inserting an item in such a priority queue requires traversing the list from front to back to find the place to insert the item, in order to keep the list sorted
  - with N items has average- and worst-case time cost $O(\quad)$

- Deleting the highest priority item in such a priority queue just requires unlinking the first item in the list
  - with N items has best-, average-, and worst-case time cost $O(\quad)$

Analysis of Huffman's algorithm with linked-list priority queue

- Suppose the input source is a sequence of length K drawn from an alphabet of N symbols

Step **0**: time cost $O(K)$

Step **1**: $O(N^2)$, or $O(N \log N)$ if counts are sorted first

Step **2**: the loop is executed N-1 times

Step **2a**: time cost $O(1)$

Step **2b**: time cost $O(1)$

Step **2c**: time cost $O(N)$

Step **3**: time cost $O(1)$

- So overall time cost: $O(K + N^2)$

- You can do better using a heap, instead of a linked list, to implement the priority queue

Implementing a priority queue using a heap

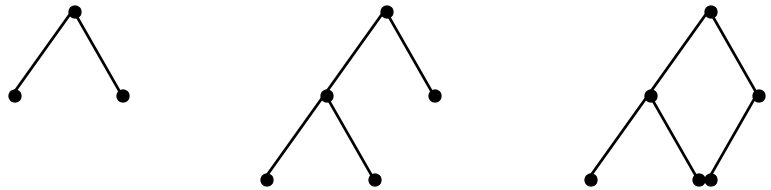
- A *heap* is a binary tree with these properties:
  - "structural property": each level is completely filled, with the possible exception of the last level, which is filled left-to-right

    (heaps are "complete" binary trees, and so are balanced: Height H = O(logN))
  - "ordering property": for each node X in the heap, the key value stored in X is greater than or equal to all key values in descendants of X

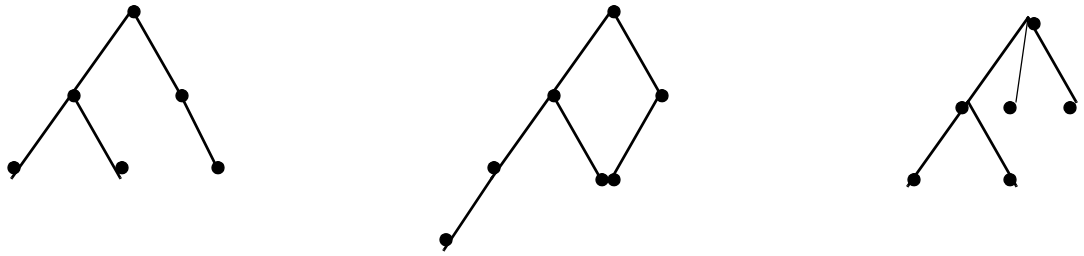
    (this is sometimes called a MAX-heap; for MIN-heaps, replace greater than with less than)
- Heaps are often just called "priority queues", because they are natural structures for implementing the Priority Queue ADT. They are also used to implement the heapsort sorting algorithm, which is a nice fast N log N sorting algorithm
- Note: This heap data structure has NOTHING to do with the "heap" region of machine memory where dynamic data is allocated...

The heap structural property

- These trees satisfy the structural property of heaps:

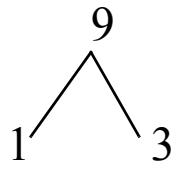


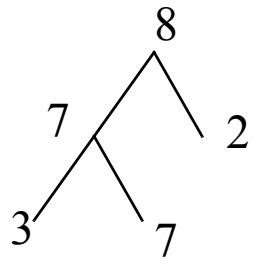
- These do not:



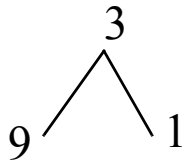
The heap ordering property

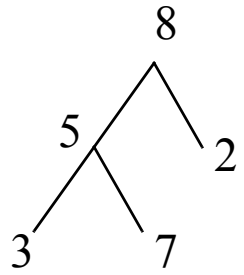
- These trees satisfy the ordering property of (max) heaps:





- These do not:

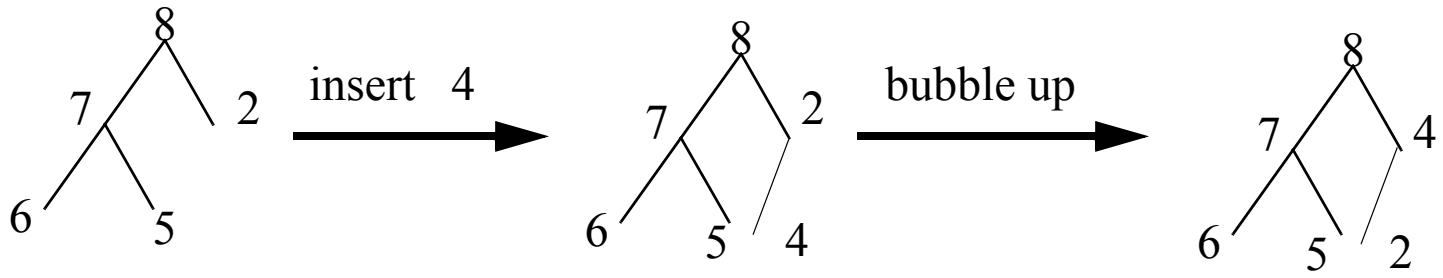




Inserting a key in a heap

- When inserting a key in a heap, must be careful to preserve the structural and ordering properties. The result must be a heap!
- Basic algorithm:

  1. Create a new node in the proper location, filling level left-to-right
  2. Put key to be inserted in this new node
  3. "Bubble up" the key toward the root, exchanging with key in parent, until it is in a node whose parent has a larger key (or it is in the root)

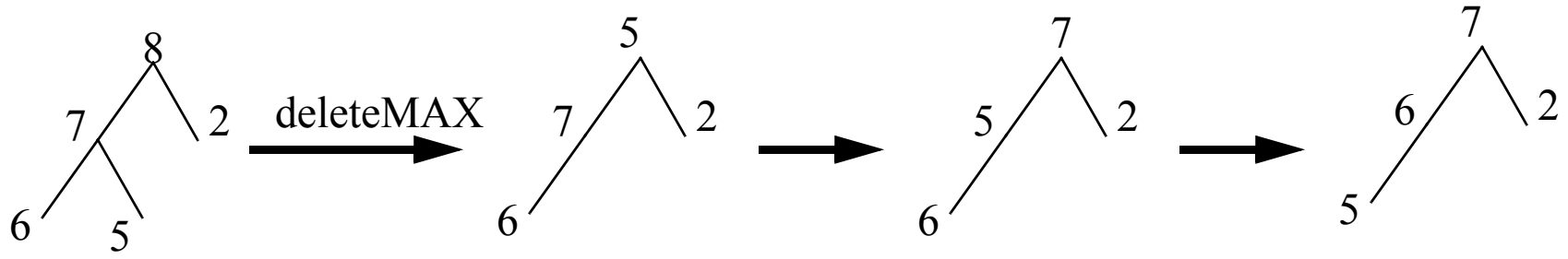


- Insert in a heap with N nodes has time cost $O(\log N)$

Deleting the maximum key in a heap

- When deleting the maximum valued key from a heap, must be careful to preserve the structural and ordering properties. The result must be a heap!
- Basic algorithm:

1. The key in the root is the maximum key. Save its value to return
2. Identify the node to be deleted, unfilling bottom level right-to-left
3. Move the key in the node to be deleted to the root, and delete the node
3. "Trickle down" the key in the root toward the leaves, exchanging with key in the larger child, until it is in a node whose children have smaller keys, or it is in a leaf



- deleteMAX in a MAX heap is $O(\log N)$. (What is deleteMIN in a MAX heap?)

Analysis of Huffman's algorithm with heap priority queue

- Suppose the input source is a sequence of length K drawn from an alphabet of N symbols

Step **0**: time cost O(K)

Step **1**: O(N log N) (can actually be done in O(N) if you are a bit clever)

Step **2**: the loop is executed N-1 times

Step **2a**: time cost O(logN)

Step **2b**: time cost O(1)

Step **2c**: time cost O(log N)

Step **3**: time cost O(1)

- So overall time cost: O(K + N log N)

- But if K is large, or slow I/O is involved, Step 0 will dominate and choice of priority queue implementation may not matter significantly

Time cost of Huffman coding

- We have analyzed the time cost of constructing the Huffman code tree
- What is the time cost of then using that tree to code the input message?
- Coding one symbol from the input message takes time proportional to the number of bits in the code for that symbol; and so the total time cost for coding the entire message is proportional to the total number of bits in the coded version of the message
- If there are K symbols in the input message, and the average number of bits per symbol in the Huffman code is H', then the time cost for coding the message is proportional to $H' \cdot K$
- What can we say about H' ? This requires a little information theory...

Sources of information and entropy

- A source of information emits a sequence of symbols drawn independently from some alphabet
- Suppose the alphabet is the set of symbols $\{\sigma_1, \ldots, \sigma_N\}$
- Suppose the probability of symbol σ_i occurring in the source is p_i
- Then the information contained in symbol σ_i is $\log\frac{1}{p_i}$ bits, and the average information per symbol is (logs are base 2):

$$H = \sum_{i=1}^{N} p_i \log\frac{1}{p_i}$$

- This quantity H is the "entropy" or "Shannon information" of the information source

- For example, suppose a source uses 3 symbols, which occur with probabities 1/3, 1/4, 5/12
- The entropy of this source is

$$\frac{1}{3}\log 3 + \frac{1}{4}\log 4 + \frac{5}{12}\log\frac{12}{5} \approx 1.5546 \text{bits}$$

Entropy and coding sources of information

- A uniquely decodable code for an information source is a function that assigns to each source symbol σ_i a unique sequence of bits, called the code for σ_i
- Suppose the number of bits in the code for symbol σ_i is n_i
- Then the average number of bits in the code for the information source is

$$H' = \sum_{i=1}^{N} p_i n_i$$

- Let H'_{min} be the average number of bits in the code with the smallest average number of bits of all uniquely decodable codes for an information source with entropy H. Then you can prove the following interesting results:
  - $H \leq H'_{min} < H + 1$, i.e. you can do no better than H, and don't need to do worse by more than one bit per symbol
  - The Huffman code for this source has codes with average number of bits H'_{min}, i.e., there is no better uniquely decodable code than Huffman

- So Huffman-coding a message consisting of a sequence of K symbols obeying the probabilities of an information source with entropy H takes at least $H \cdot K$ bits but less than $H \cdot K + K$ bits

How large and how small can entropy be?

- A source of information emits a sequence of symbols drawn independently from the alphabet $\{\sigma_1, \ldots, \sigma_N\}$ such that the probability of symbol σ_i occurring is p_i
- The entropy (Shannon information) of the source, in bits, is defined as (logs are base 2):

$$H = \sum_{i=1}^{N} p_i \log \frac{1}{p_i}$$

- Q: What is the possible range of values of H? A: We always have $0 \le H \le \log N$
- The smallest possible value of H is 0:
  - If one symbol σ_i occurs all the time, so $p_i = 1$ and so $\log 1/p_i = 0$, and all the other symbols σ_j never occur, so the other $p_j = 0$, then you don't get any information by observing the source:

$$H = 0$$

- The largest possible value of H is log$N$. This is the 'maximum entropy' condition
  - If each of the symbols are equally likely, then $p_i = 1/N$ for all i and so:

$$H = \sum_{i=1}^{N} \frac{1}{N} \log N = \log N$$

Entropy and Huffman: an example

- Suppose the alphabet consists of the 8 symbols A,B,C,D,E,F,G,H, and the message sequence to be coded is AAAAABBAHHBCBGCCC
- Given an alphabet of 8 symbols, the entropy H of a source using those symbols must lie in the range $0 \le H \le \log_2 N = 3$. But if the source emits symbols with probabilities as evidenced in this message, what exactly is its entropy H?
- The table of counts or frequencies of symbols in this message would be:
  A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2
- The message has length 17, and so the probabilities of these symbols are:
  A: 6/17; B: 4/17; C: 4/17; D: 0; E: 0; F: 0; G: 1/17; H: 2/17
- The entropy of this message (and any message with those symbol probabilites) is then

$$\frac{6}{17}\log\frac{17}{6} + \frac{4}{17}\log\frac{17}{4} + \frac{4}{17}\log\frac{17}{4} + 0 + 0 + 0 + \frac{1}{17}\log 17 + \frac{2}{17}\log\frac{17}{2} \approx 2.1163 \text{ bits}$$

- As we saw last time, the Huffman code for this message gives $H'_{min} = 37/17 \approx 2.1768$ bits per symbol, which is quite close to the message entropy

Huffman code trees, using dynamic data and pointers

- A class definition for a Huffman code tree node might have member variables along these lines (assume the symbols to be coded come from an alphabet of no more than 256 items):

```
class HCNode {
   HCNode* parent;     // pointer to parent; null if root
   bool isChild0;      // true if this is "0" child of its parent
   HCNode* child0;     // pointer to "0" child; null if leaf
   HCNode* child1;     // pointer to "1" child; null if leaf
   unsigned char symb;    // symbol
   int count;          // count/frequency of symbols in subtree
};
```

- The class definition could also have some methods and constructors for common operations and initialization of `HCNode` objects
- When building a Huffman code tree, the fields of the parent and child nodes need to be set appropriately
- The result is a tree data structure that is useful both for coding and decoding
- (For coding, you should also have an array or other table structure of pointers to code tree leaf nodes, indexed by data items to code)

Implementing trees using arrays

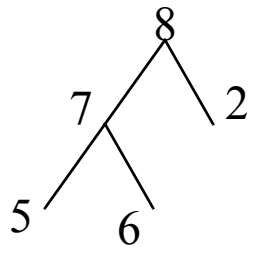
- Arrays can be used to implement trees; this is a good choice in some cases if the tree has a very regular structure, or a fixed size
- One common case is: implementing a heap
- Because heaps have a very regular structure (they are complete binary trees), the array representation can be very compact: array entries themselves don't need to hold parent-child relational information
  - the index of the parent, left child, or right child of any node in the array can be found by simple computations on that node's index
- Huffman code trees do not have as regular a structure as a heap (they can be extremely unbalanced), but they do have some structure, and they do have a determinable size
  - structure: they are "full" binary trees; every node is either a leaf, or has 2 children
  - size: to code N possible items, they have N leaves, and N-1 internal nodes
- These features make it potentially interesting to use arrays to implement a Huffman code tree

Implementing heaps using arrays

- The heap structural property permits a neat array-based implementation of a heap
- Recall that in a heap each level is filled left-to-right, and each level is completely filled before the next level is begun (heaps are "complete" binary trees)
- One way to implement a heap with N nodes holding keys of type T, is to use an N-element array `T heap[N];` Nodes of the heap will correspond to entries in the array as follows:
  - The root of the heap is array element indexed 0, `heap[0]`
  - if a heap node corresponds to array element indexed `i`, then
    - its left child corresponds to element indexed `2*i + 1`
    - its right child corresponds to element indexed `2*i + 2`
    - ..and so a node indexed `k` has parent indexed `(k-1)/2`
- The result: Nodes at the same level are contiguous in the array, and the array has no "gaps"
- (Note: Weiss shows a slightly different scheme, with an array of length N+1)
- Now it is easy to implement the heap Insert and Delete operations (in terms of "bubbling up" and "trickling down"); and easy to implement O(N logN) “heap sort”, in place, in a N-element array

Implementing heaps using arrays: a picture

- To map out the correspondence between a tree-diagram and array version of a heap, follow the formulas, or follow the rule for the structural property of a heap: levels are filled left to right, and top to bottom



8	7	2	5	6		
0	1	2	3	4	5	6

Priority queues in C++

- It is not too hard to code up your own priority queue, using the heap-structured array
- Also, the C++ STL has a `priority_queue` class template, which provides functions `push()`, `top()`, `pop()`, `size()`
- 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;
```

- However, objects in a priority queue must be comparable to each other for priority
- 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`
- So let's see how to override that operator for `HCNode`s

Overriding operator< for HCNodes: header file

- In a header file, say HCNode.hpp:

```
#ifndef HCNODE_HPP
#define HCNODE_HPP
class HCNode {

   friend class HCTree; // so an HCTree can access HCNode fields

private:
   HCNode* parent;     // pointer to parent; null if root
   bool isChild0;      // true if this is "0" child of its parent
   HCNode* child0;     // pointer to "0" child; null if leaf
   HCNode* child1;     // pointer to "1" child; null if leaf
   unsigned char symb;   // symbol
   int count;           // count/frequency of symbols in subtree

public:
   // for less-than comparisons between HCNodes
   bool operator<(HCNode const &) const;
};
#endif
```

Overriding operator< for HCNodes: implementation file

- In an implementation file, say HCNode.cpp:

```
#include "HCNode.hpp"

/** Compare this HCNode and other for priority ordering.
  * Smaller count means higher priority.
  * Use node symbol for deterministic tiebreaking
  */
bool HCNode::operator<(HCNode const & other) const {
   // if counts are different, just compare counts
   if(count != other.count) return count > other.count;
   // counts are equal.  use symbol value to break tie.
   // (for this to work, internal HCNodes must have symb set.)
   return symb < other.symb;
}
```

Using operator< to compare HCNodes

- Consider this context:

```
HCNode n1, n2, n3, n4;
n1.count = 100; n1.symb = 'A';
n2.count = 200; n2.symb = 'B';
n3.count = 100; n3.symb = 'C';
n4.count = 100; n4.symb = 'A';
```

- Now what is the value of these expressions?

```
n1 < n2

n2 < n1

n2 < n3

n1 < n3

n3 < n1

n1 < n4
```

Using std::priority_queue in Huffman's algorithm

- If you create an STL container such as priority_queue to hold HCNode objects:

```
#include <queue>

std::priority_queue<HCNode> pq;
```

- ... then adding an HCNode object to the priority_queue:

```
HCNode n;
pq.push(n);
```

- ... actually creates a *copy* of the HCNode, and adds the copy to the queue. You probably don't want that. Instead, set up the container to hold pointers to HCNode objects:

```
std::priority_queue<HCNode*> pq;
HCNode* p = new HCNode();
pq.push(p);
```

- But there's a problem: our `operator<` is a member function of the `HCNode` class. It is not defined for pointers to HCNodes. What to do?

std::priority_queue template arguments

- The template for priority_queue takes 3 arguments:

```
template < class T,
           class Container = vector<T>,
           class Compare = less<typename 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,
  - operator< is used for priority comparisons

- Using a vector for the internal store is fine, but we want to tell the priority_queue to first dereference the HCNode pointers it contains, and then apply operator<

- How to do that?

Defining a "comparison 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 strict weak ordering operation. This can be a class implementing a function call operator...
- Here's how to define a class implementing the function call operator `operator()` that performs the required comparison:

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

- Now, create the priority_queue as:

```
std::priority_queue<HCNode*,std::vector<HCNode*>,HCNodePtrComp> pq;
```

- and priority comparisons will be done as appropriate.

Huffman code trees using arrays: one approach

- Suppose, for example, that the items to be coded are bytes: 8-bit integers
- A Huffman code tree for this domain will have (no more than) 256 leaves, and (no more than) 255 internal nodes

- We could represent the tree using arrays along these lines:

```
// this array will hold indices of parents of leaf nodes
unsigned char leaves[256];
// this array will hold indices of parents of internal nodes
unsigned char intnodes[255];
```

- Now given the value of an unsigned char `c` to code, `leaves[c]` gives the index of its parent in the `intnodes` array
- ... `intnodes[leaves[c]]` gives the index of the parent of that node in the `intnodes` array, etc.
- (Note the `unsigned char` type in C++ has values in the range 0 through 255 inclusive)
- The root of the tree can be indicated by an array entry with value 255 (its parent is not in the `intnodes` array...)

“0” child or “1” child?

- That array representation will indicate which nodes are the children of which other nodes in the tree...
  - But it does not indicate whether a child is a “0” child or a “1” child, and this is essential for the coding tree
- We could follow a convention such as: of the two children, the one with smaller index is always the “1” child (and all leaf node indices are considered smaller than all internal node indices)
- ... Or, we can represent these additional “bits” of information about the nodes using two bit sequences: one with 256 bits for the leaf nodes, and one with 255 bits for the internal nodes

- These bit sequences can be packed into 32 bytes each:

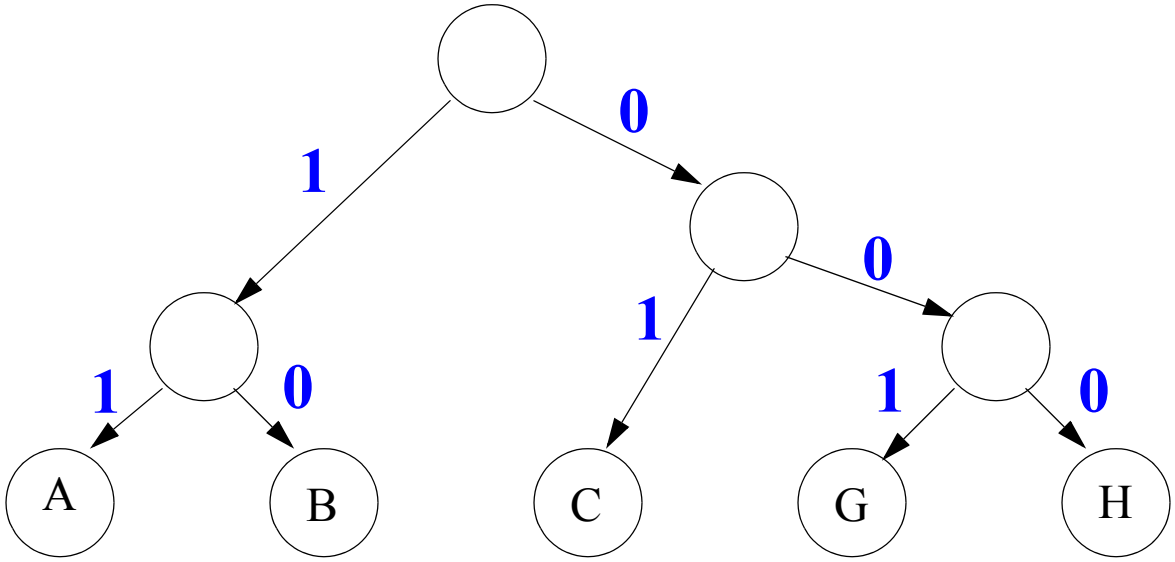
```
// this array holds bits indicating what kind of child a leaf is
char leafchildbit[32];
// this array holds bits indicating what kind of child a nonleaf is
char intnodechildbit[32];
```

- For example: if bit 65 is “1” in the `leafchildbit` array, that indicates that the leaf node with data value 65 is a “1” child of its parent

Huffman code trees using arrays, cont'd

- Let's see how these arrays would look when representing the Huffman code tree we have been using as an example
- Note that the ASCII code values of characters A,B,C,G,H are 65,66,67,71,72 respectively
- See picture on next page...

A picture: an array representation of a Huffman tree



leaves

...	2	2	1				0	0	...
...	65	66	67	68	69	70	71	72	...

intnodes

1	3	3	255			
0	1	2	3	...		

leafchildbit: bits 65,67,71 are 1, others 0

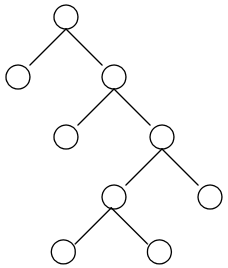
intnodechildbit: bit 2 is 1, others 0

About this array representation

- This array representation scheme faithfully and completely represents the Huffman code tree, and is useful as a coding tree (easy to follow links from leaf to root)
- But the scheme does not explicitly indicate for a given node what the children of that node are... (that information is implicit, not explicit, in that scheme)
- To be useful as a tree for efficient decoding, the representation could be augmented with arrays that explicitly indicate, for each node, what nodes are its "1" and "0" children, together with an explicit index of the root node; or it could be converted into such a representation
- However, as it is, this scheme is a compact representation of the entire tree; it permits representing any Huffman code on a 256-symbol alphabet using only 256 + 255 + 32 + 32 = 575 bytes
- Recall that enough information to reconstruct the Huffman coding tree should be stored in a Huffman-compressed file; these arrays (or a variant of them) would be one way of doing that

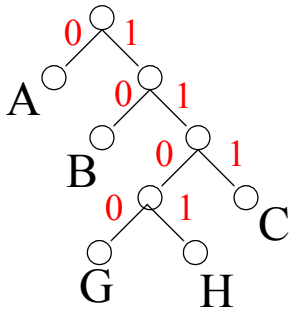
Huffman code trees using arrays: another approach

- The approach just sketched made no use of the special structure of Huffman code trees, other than the fact that if there are N leaf nodes, there are N-1 internal nodes
- Using the fact that the Huffman tree is “full” -- every node is either a leaf or has 2 children -- permits another approach leading to a more compact representation
- Consider a usual (inorder or preorder or postorder) traversal of a binary tree: it visits the leaves in left-to-right order
- Suppose when visiting each leaf, its level in the tree is recorded
- This sequence of left-to-right leaf levels does not specify the structure of the tree for arbitrary binary trees... but it does for full binary trees!
- For example, the left-to-right leaf level sequence 1,2,4,4,3 can be obtained only from this full binary tree:



Huffman code trees using arrays: another approach, cont'd

- So, you can uniquely reconstruct a Huffman code trie with the following information:
  - the left-to-right sequence of leaf levels
  - the corresponding sequence of message symbols in those leaves
  - ... together with an assumption about which (left or right) children are "0" children
- For example: the level sequence 1,2,4,4,3 and the symbol sequence A,B,G,H,C and the assumption that left children are "0" children uniquely determine this Huffman trie:



- Assuming a 256-symbol alphabet, leaf levels are in the range 0-255, and so with this approach any code trie could be represented using at most 256 + 256 bytes; or by using a byte to specify the value *m* equal to how many of the 256 possible symbols actually appear in the message, the trie could be represented using $1 + 2m$ bytes
- Note that this representation isn't directly useful for coding or decoding... you would use it to reconstruct the tree, and then use the tree

Representing the information in a Huffman code tree

- Huffman's algorithm does construct a binary trie, which represents a coding scheme to use for coding and decoding messages
- But for actually coding and decoding a message, it really isn't required to use a binary trie
- What you need is:
  - a way to go from message symbols to bit sequences, for coding a message
  - a way to go from bit sequences to message symbols, for decoding a message
- As we have seen, the Huffman code tree itself permits doing those things, but other data structures also can
- For example:
  - An array of Strings of "1"'s and "0"s, indexed by message symbol
  - A HashMap in which keys are Strings of "1"'s and "0"s, and values are message symbols
- Other approaches are possible as well. "In computer science, if there is one way to do something, there are infinitely many ways to do it!"

Next time...

- Overview of C++ I/O
- Some useful classes in <iostream>
- I/O buffering
- Bit-by-bit I/O

Reading: online documentation on C++ streams