CSE 100: HUFFMAN CODES AND C++ IO

Q1: Which of the following is the best choice of data structure for storing the forest of trees in the construction of the Huffman Tree?

A. BST
B. Sorted array
C. Linked-list
D. Heap

Q2: Which class do you use to read from a file in C++?

A. cin
B. ifstream
C. ofstream
D. file

Q3: What is the smallest amount of data you can read from a file using the C++ defined methods?

A. A bit
B. A byte
C. A word
D. It depends on the operating system

Q4: Of the following, which method would you use to read from a binary file in C++, without loss of information?

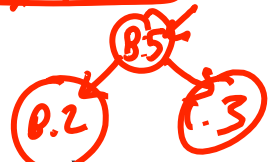
A. getline
B. The << operator
C. The >> operator
D. read

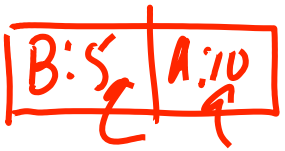
Huffman's algorithm: Building the Huffman Tree

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.

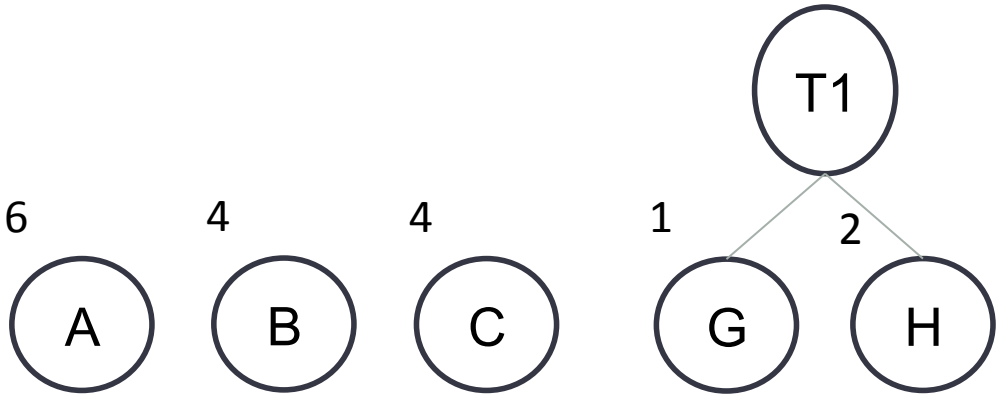
2. Loop while there is more than 1 tree in the forest:
   2a. Remove the two lowest count trees
   2b. Combine these two trees into a new tree (summing their counts).
   2c. Insert this new tree in the forest, and go to 2.





3. Return the one tree in the forest as the Huffman code tree.

Huffman Algorithm: Forest of Trees



Delete-min

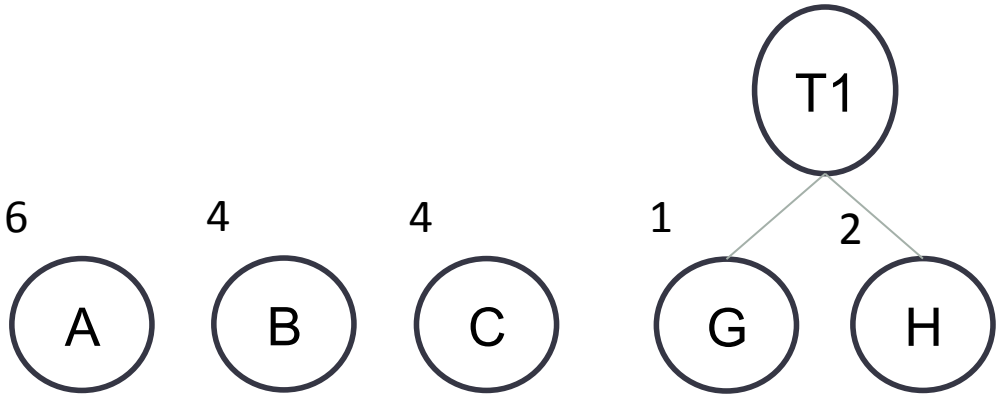
Insert

What is a good data structure to use to hold the forest of trees?

A. BST (If balanced) or $\log_2 N$) $O(\log_2 N)$
B. Sorted array $O(N)$ $O(N)$
C. Linked list $O(N)$ $O(1)$
D. Something else or $\log_2 N$) or $\log_2 N$)

Heap

Huffman Algorithm: Forest of Trees



What is a good data structure to use to hold the forest of trees?

A. BST: Supports min, insert and delete in O(log N)
B. Sorted array: Not good for dynamic data
C. Linked list: If unordered then good for insert (constant time) but min would be O(N). If ordered then delete, min are constant time but insert would be O(N)
D. Something else: Heap (new data structure?)

What is a Heap?

In priority queue, Root should have:
A. lowest count
B. highest count

Think of a Heap as a binary tree that is as complete as possible and satisfies the following property:

min-Heap

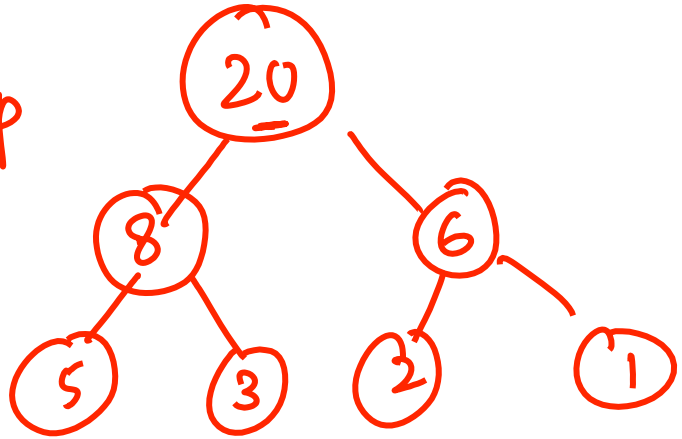
```
At every node x
  Key[x]<= Key[children of x]
```

key[x] ≥ key[children] → condition for max-heap

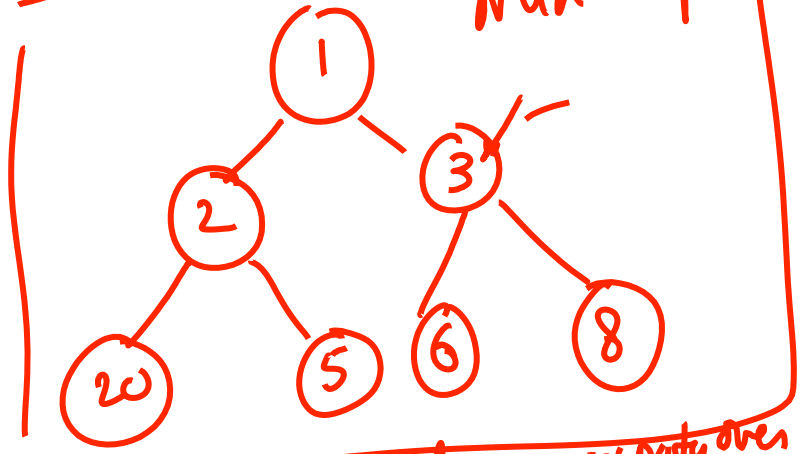
std::priority_queue<HCNode> Q

So the root has the min value in a min heap

Max-heap



Min-heap



In priority-queue: HCNode n1, n2;

n1 < n2

If true, n1 has lower priority over n2

Heap vs. BST vs. Sorted Array

Operations	BST (Balanced)	Sorted Array	Heap
Search	O(log N)	O(log N)	X
Selection	O(log N)	O(1)	X
Min and Max	O(log N)	O(1)	X
Min or Max	O(log N)	O(1)	O(log₂N)
Predecessor/ Successor	O(log N)	O(1)	X
Rank	O(log N)	O(log N)	X
Output in sorted order	O(N)	O(N)	X
Insert	O(log N)	O(N)	O(log₂N)
Delete	O(log N)	O(N)	X
Extract min or extract max	O(log₂N)	O(N)	O(log₂N)

Ref: Tim Roughgarden (Stanford)

The suitability of Heap for our problem

- In the Huffman problem we are doing repeated inserts and extract-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

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)

Priority Queues in C++

```
#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`**

Priority Queues in C++

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

public:
  HCNode* parent; // pointer to parent; null if root
  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

  // for less-than comparisons between HCNodes
  bool operator<(HCNode const &) const;
};

#endif
```

need to overload this operator so that nodes with lower count have higher priority

In HCNode.cpp:

```
#include HCNODE_HPP
/** Compare this HCNode and other for priority
ordering.
 * Smaller count means higher priority.
 */
bool HCNode::operator<(HCNode const & other) const {
  // if counts are different, just compare counts
  return count > other.count;
};

#endif
```

What is wrong with this implementation?

A. Nothing
B. It is non-deterministic (in our algorithm)
C. It returns the opposite of the desired value for our purpose

In 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;
};

#endif
```

Using < to compare nodes

- 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   false
n2 < n1   true
n2 < n3   true
n1 < n3   true
n3 < n1   false
n1 < n4   false
```

A. true
B. false

Using std::priority_queue in Huffman's algo

- 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);
```

Using std::priority_queue in Huffman's

Instead, set up the container to hold pointers to HCNode objects:

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

What is the problem with the above approach?

A. Since the priority queue is storing copies of HCNode objects, we have a memory leak

B. The nodes in the priority queue cannot be correctly compared

C. The node is created on the run time stack rather than the heap

Using std::priority_queue in Huffman's algo

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

What is the problem with the above approach?

- 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:

```
1 template < class T, class Container = vector<T>,
2            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,
  - **`less`** a class that provides priority comparisons
- Okay to use vector container
- But we need to provide the priority_queue with a Compare class

Defining a "comparison class"

- The prototype for priority_queue:

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

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

```
std::priority_queue<HCNode*> pq;
```
// Priority queue using the default less than operator

```
std::priority_queue<HCNode*,std::vector<HCNode*>,HCNodePtrComp> pq;
```
// Priority queue using our Compare class

Defining a "comparison class"

- The prototype for priority_queue:

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

- 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; }
};
```

overloaded less than.

- Now, we create the priority_queue, and priority comparisons will be done as needed

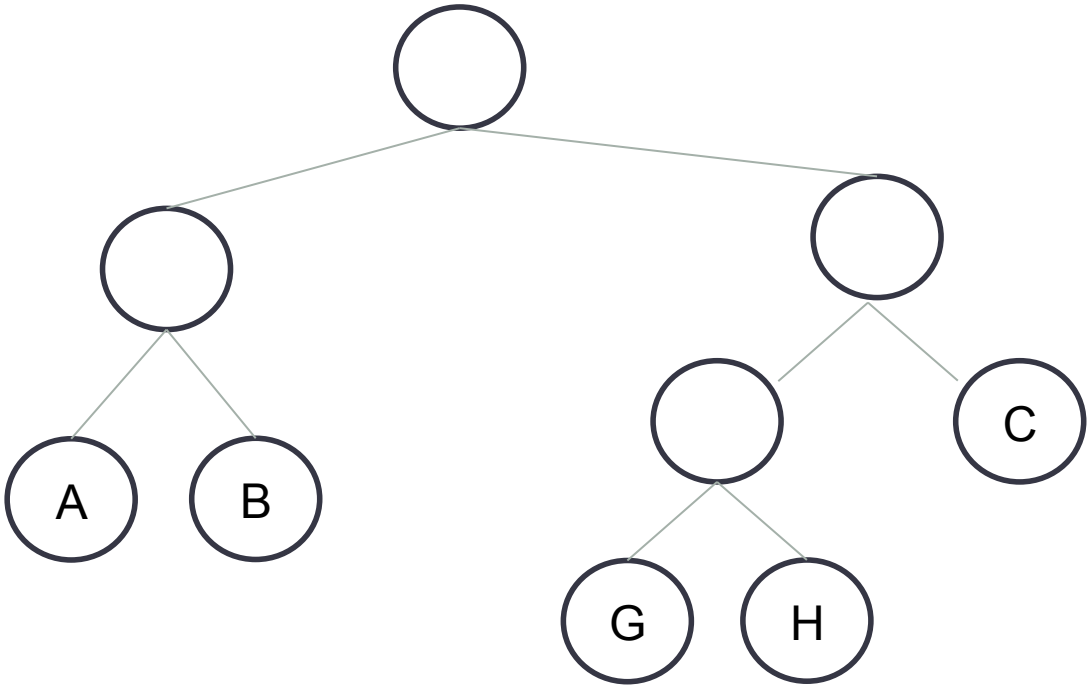
```
std::priority_queue<HCNode*,std::vector<HCNode*>,HCNodePtrComp> pq;
```

- Here is how a comparison will be done inside priority_queue

```
HCNodePtrComp nodeComparator;
If (nodeComparator(pnode1, pnode2) ) { ….}
```

- We have defined operator < on HCNode, to perform the comparison

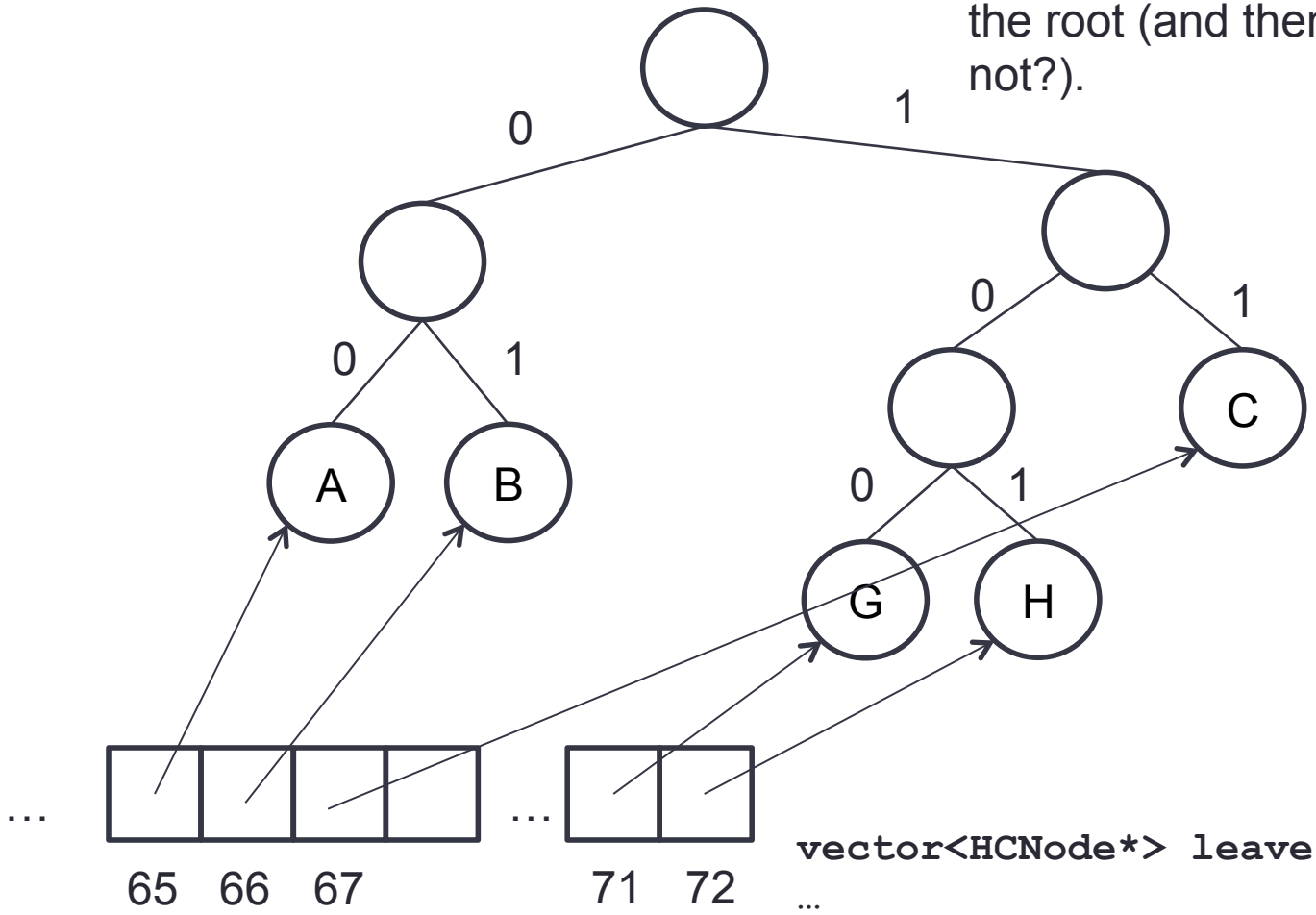
Encoding a symbol



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.

Encoding a symbol

A much better way is to maintain a list of leaves and then to traverse the tree to the root (and then reverse the code… or not?).




```
vector<HCNode*> leaves;
…
leaves = vector<HCNode*>(256, (HCNode*)0);
```

Traversing a list

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

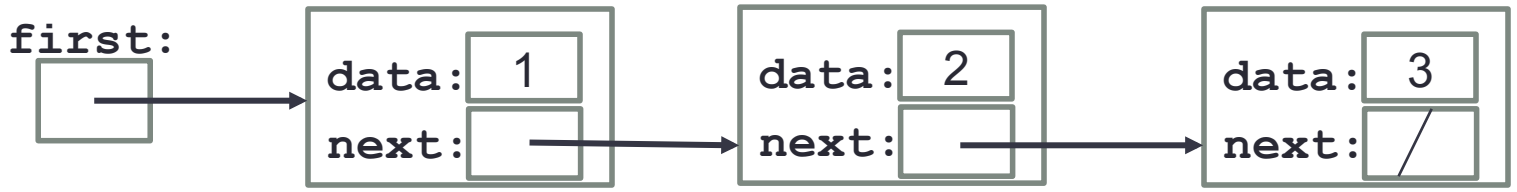
What does `traverse(first)` print?

A. 1 2 3

B. 3 2 1

C. Something else

Assume you have created the following list:




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

Traversing a list, with recursion

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


```
void traverse(LNode* n) {
  // 1
  if (n == 0) return;
  // 2
  traverse(n->next);
  // 3
}
```

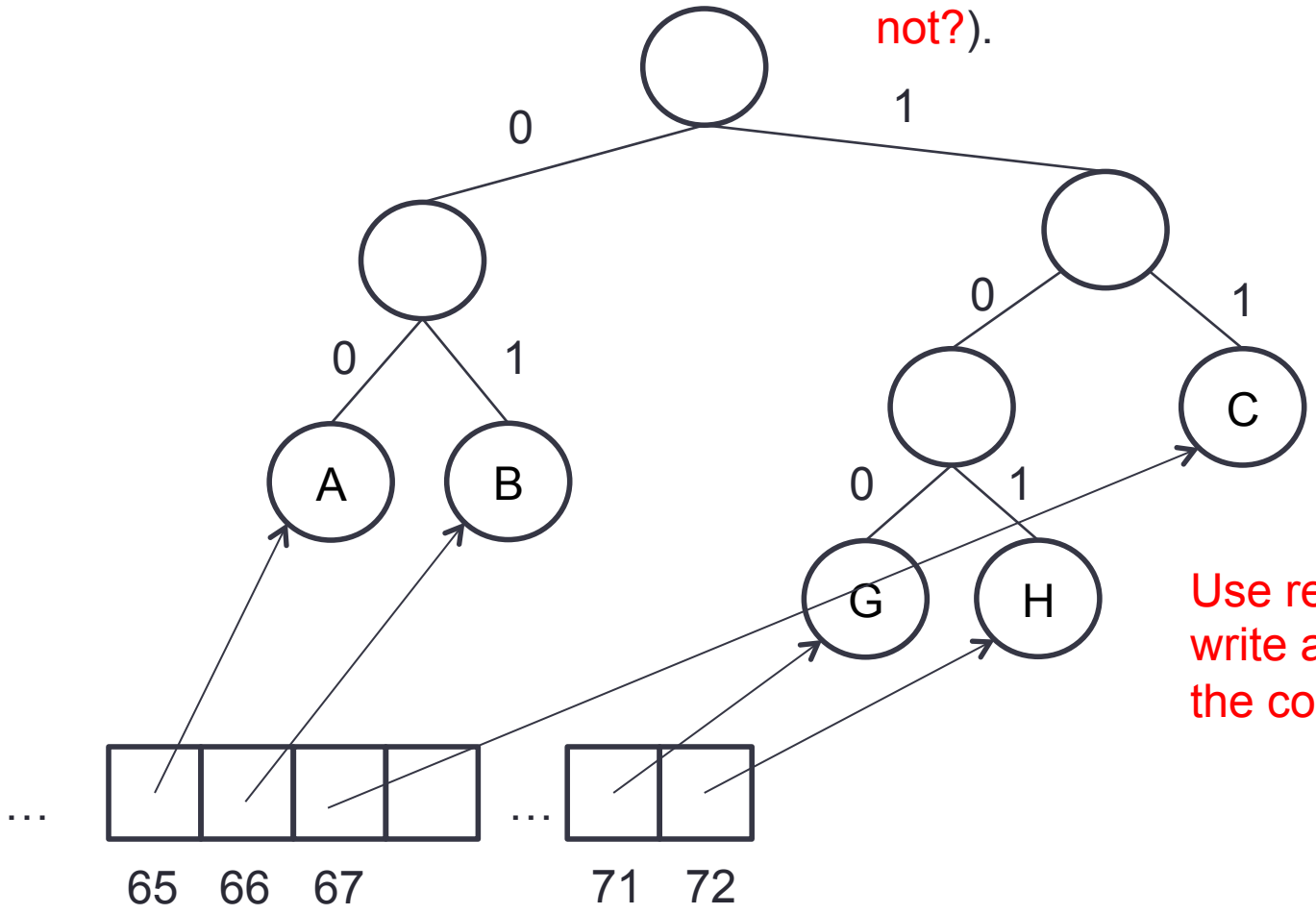
Where should I put the line to print `n->data` to print the list in *reverse order*?

```
std::cout << n->data << std::endl;
```

A. 1 B. 2 C. 3

Encoding a symbol

A much better way is to maintain a list of leaves and then to traverse the tree to the root (and then reverse the code… or not?).



Use recursion to easily write a symbol's code in the correct order!

```
vector<HCNode*> leaves;
…
leaves = vector<HCNode*>(256, (HCNode*)0);
```

PA 2 Implementation strategy

- Implement Huffman tree build() method
  - HCNode.cpp and HCTree.cpp
- Write verification code to check that you can construct simple Huffman trees correctly
  - Use small inputs that you can verify by hand
  - Output codes as strings of 1s and 0s (char)
- Write the encode and decode method
  - Test with simple inputs that you can verify by hand and output the encoded input as character strings of 1s ad 0s

  CHECKPOINT HERE!
- Add binary I/O
  - Write implementations of BitInputStream and BitOutputStream that write/ read the compressed file as a binary files \
- Compress/decompress a small file (100 bytes)
- Decompression should map the encoded input back to the original input

Huffman: Encode & Decode File I/O

Encode
(compress)

Decode
(uncompress)

- C++ I/O streams
- I/O buffering
- Bit-by-bit I/O

What is a stream?

Streams are essentially sequences of bytes of infinite length that are buffered.

C++ istream

- The istream class introduces member functions common to all input streams (that is, streams used for input into your program)
- Some important ones are:

cin is an instance of istream

istream& operator>> (type & val);

- This is the stream extraction operator, overloaded for many primitive types type
  Performs an input operation on an istream generally involving some sort of interpretation of the data (like translating a sequence of numerical characters to a value of a given numerical type)
  Returns a reference to the istream, so extractions can be 'chained'

```
std::cin >>i>> j;
```

int get();

- Perform basic unformatted input. Extracts a single byte from the stream and returns its value (cast to an int)

```
int k = cin.get( );
```

istream& read (char* s, streamsize n);

- Perform unformatted input on a block of data. Reads a block of data of n bytes and stores it in the array pointed to by s

```
char buff[40];
cin.read((buff,30);
```

C++ ostream

- The **ostream** class introduces member functions common to all output streams (streams used for output from your program)
- Some important ones are:

cout and cerr are instances of ostream

```
ostream & operator<< (type & val );
```

- This is the stream insertion operator. It is overloaded for many primitive types ***type***. It performs an output operation on an ostream generally involving some formatting of the data (like for example writing a numerical value as a sequence of characters). It returns a reference to the ostream, so insertions can be 'chained'.
  std::cout << a << " and " << b << std::endl;

```
ostream & put(char c);
```

- Perform basic unformatted output. Writes a single byte to the stream and returns a reference to the stream

```
ostream & write ( const char* s , streamsize n );
```

- Perform unformatted output on a block of data. Write a block of data of **n** bytes starting at address **s**.

```
ostream & flush ( );
```

- Any unwritten characters in the ostream's buffer are written to its output destination as soon as possible ("flushed").

C++ ifstream and ofstream

- The **`ifstream`** class inherits from istream, and introduces functions specialized for doing input from files:

```
void open ( const char * filename,
            ios_base::openmode mode = ios_base::in );
```

  - Opens a file whose name is **`filename`**.

```
void close ( );
```

  - Closes the file associated with the stream. The stream is flushed first

- The **`ofstream`** class inherits from ostream and introduces functions specialized for doing output to files:

```
void open ( const char * filename,
            ios_base::openmode mode = ios_base::out );
```

  - Opens a file whose name is **`filename`**.

```
void close ( );
```

  - Closes the file associated with the stream.

Reading from a file

```
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
  ifstream theFile;
  string nextWord;
  theFile.open( "testerFile.txt" );
  while ( 1 ) {
    theFile >> nextWord;
    if (theFile.eof()) break;  // Also if (!theFile.good()) break
    cout << nextWord << " ";

  }
  theFile.close();
}
```

Identify the C++ operator that is allowing us to read from the file!

Reading from a file

```
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
  ifstream theFile;
  string nextWord;
  theFile.open( "testerFile.txt" );
  while ( 1 ) {
    theFile >> nextWord;
    if (theFile.eof()) break;  // Also if (!theFile.good()) break
    cout << nextWord << " ";

  }
  theFile.close();
}
```

Identify the C++ operator that is allowing us to read from the file!

Notice that this code will strip formatting and read whole strings! (*Not what you should do for your internal checkpoint…)*

Reading *bytes* from a file

```
#include <iostream>
#include <fstream>

using namespace std;

int main( int argc, char** argv )
{
  ifstream theFile;
  char nextChar;
  theFile.open( "testerFile.txt" );
  while ( 1 ) {
    nextChar = _____________________________;
    if (theFile.eof()) break;
    cout << nextChar;

  }
  theFile.close();
}
```

```
A. theFile.get();
B. (char)theFile.get();
C. (int)theFile.get();
D. All of the above
```

What is the difference between this approach and using the >> operator?

What should go in the blank so that we read a character at a time from a text file?

Writing to a file

- In your Huffman code program you will write the encoded text from the infile to an outfile by writing out the code (a sequence of 0s and 1s) for each character in sequence.
- What is wrong with using with the following method for writing these codes to the file?

```
// assume that outStream is an ofstream, n is an HCNode
// and HCNode has a boolean field isZeroChild
…
if (n->isZeroChild) {
  outStream << '0';
}
else {
  outStream << '1';
}
```

A. Nothing
B. You cannot use the << operator to write to a file in C++
C. The 'compressed' file will be larger than the uncompressed file
D. The bits in the code will be written in the wrong order

I/O

- In your Huffman code program you will write the encoded text from the infile to an outfile by writing out the code (a sequence of 0s and 1s) for each character in sequence.
- What is wrong with using with the following method for writing these codes to the file?

```
// assume that outStream is an ofstream, n is an HCNode
// and HCNode has a boolean field isZeroChild
…
if (n->isZeroChild) {
  outStream << '0';
}
else {
  outStream << '1';
}
```

A. Nothing
B. You cannot use the << operator to write to a file in C++
C. The 'compressed' file will be larger than the uncompressed file
D. The bits in the code will be written in the wrong order

But this is exactly what you will do for the internal Checkpoint deadline! (We'll talk about how to write one bit at a time later)