CSE 100: RBT AND HUFFMAN ALGORITHM

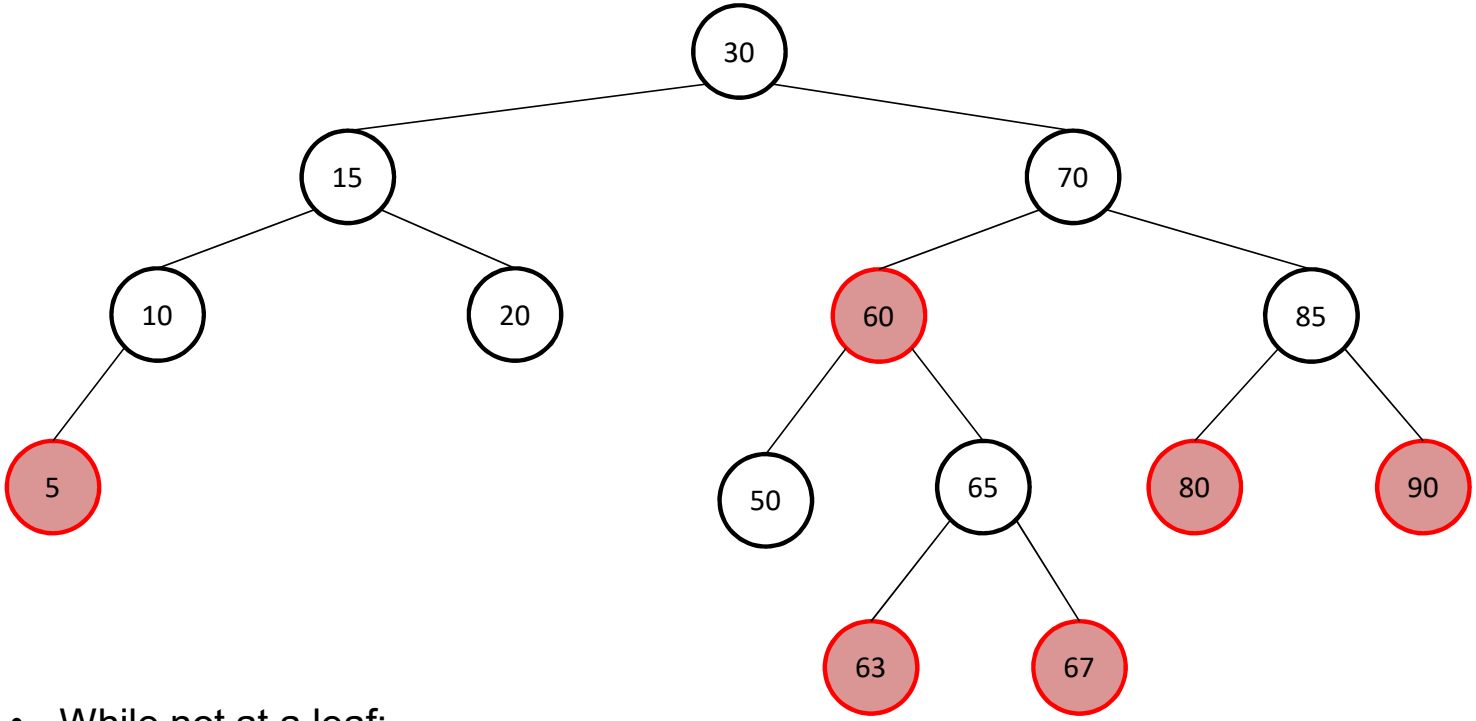
Announcements

- HW4 out
  - Homework 4 - Due 11/15 @ 11:59PM
- PA2
  - Final Deadline: 11:59pm on Tuesday, 11/13 (slip day eligible)
- Monday
  - Veterans day holiday – No class

Can any node have 2 red children?

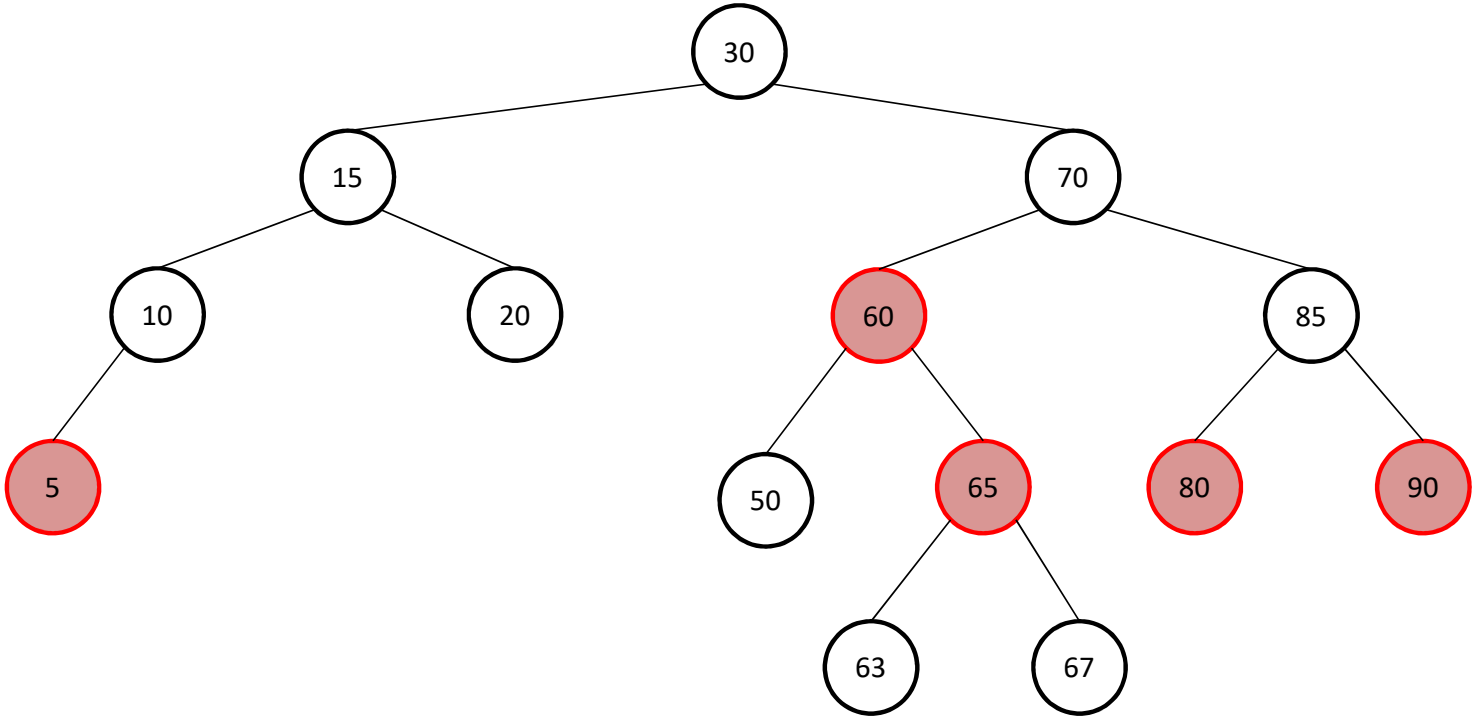
- As we descend the tree, we detect if a node X has 2 red children, and if so we do an operation to change the situation
- Note that in doing so:
  - we may change things so that a node above X now has 2 red children, where it didn't before! (example: node 60 after we insert 35)
  - if we have to do a double rotation, we will move X up and recolor it so that it becomes black, and has 2 red children itself! (example: work through inserting 64 in the tree on the following page)
- But neither of these is a problem, because
  - it never violates any of the properties of red-black trees (those 2 red nodes will always have a black parent, for example),
  - and the 2 red siblings will be too "high" in the tree for either of them to be the sibling of the parent of any red node that we find or create when we continue this descent of the tree

Exercise: Insert 64 into this tree



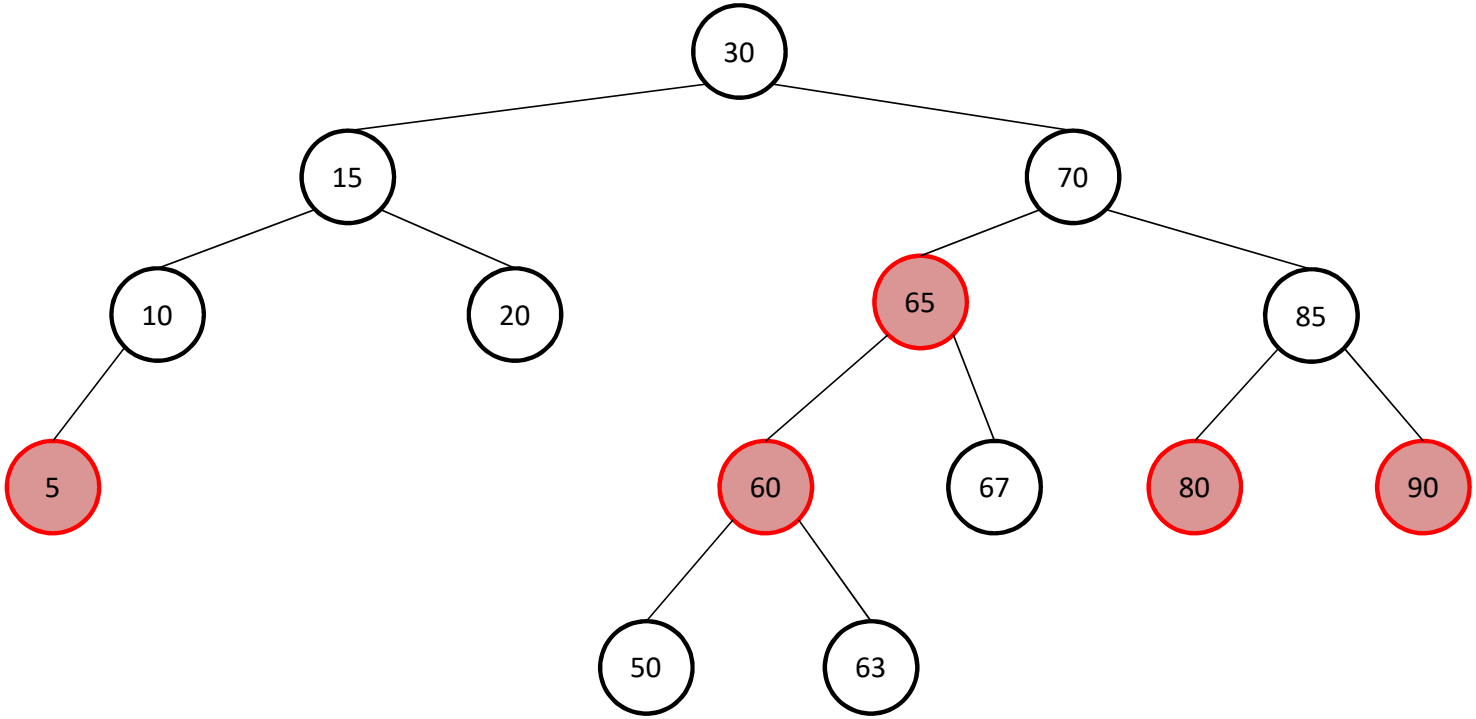
- While not at a leaf:
  - Move down the tree to where node should be placed
  - If you encounter a node with two red children, recolor, then perform any necessary rotations to fix the tree
- Insert the node
- Perform any necessary rotations to fix the tree

Exercise: Insert 64 into this tree



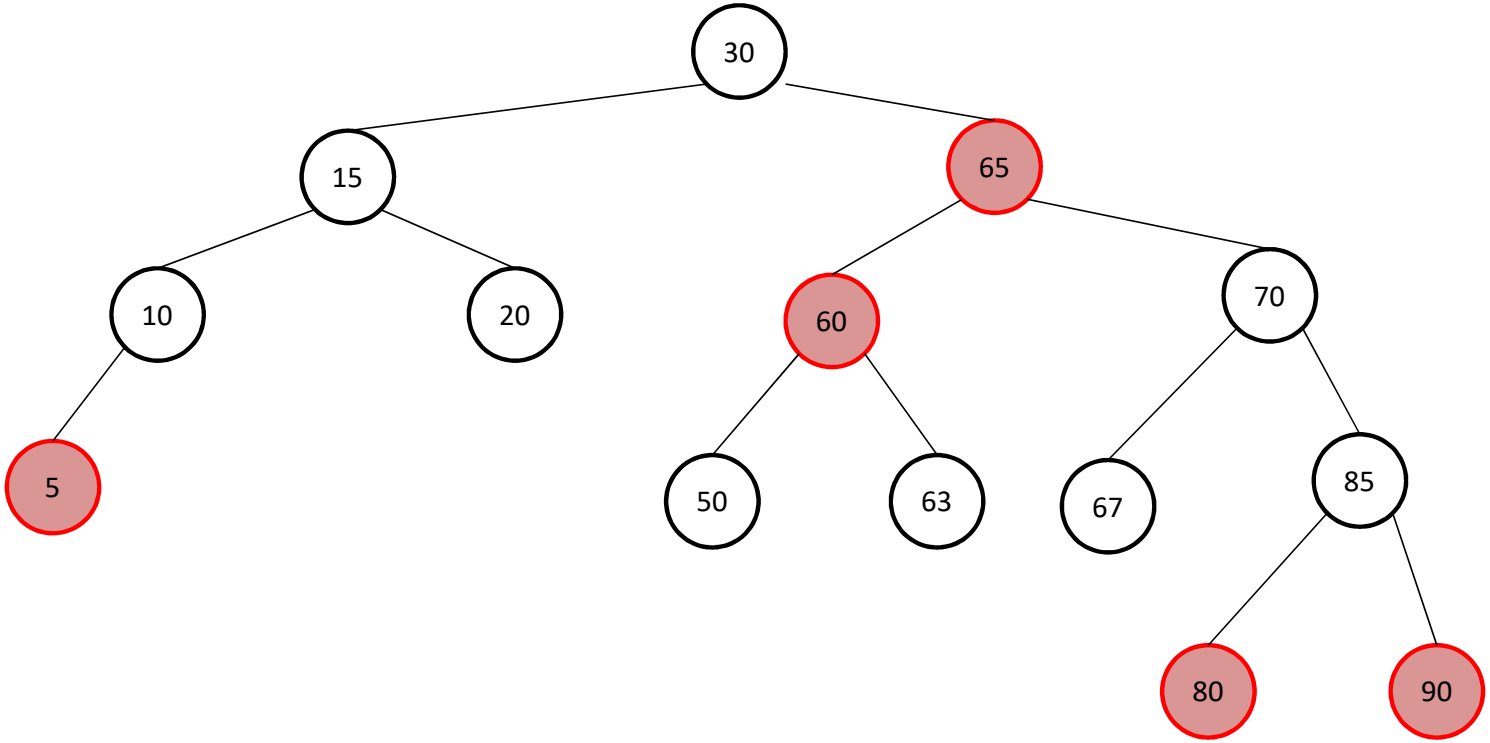
Recolor

Exercise: Insert 64 into this tree



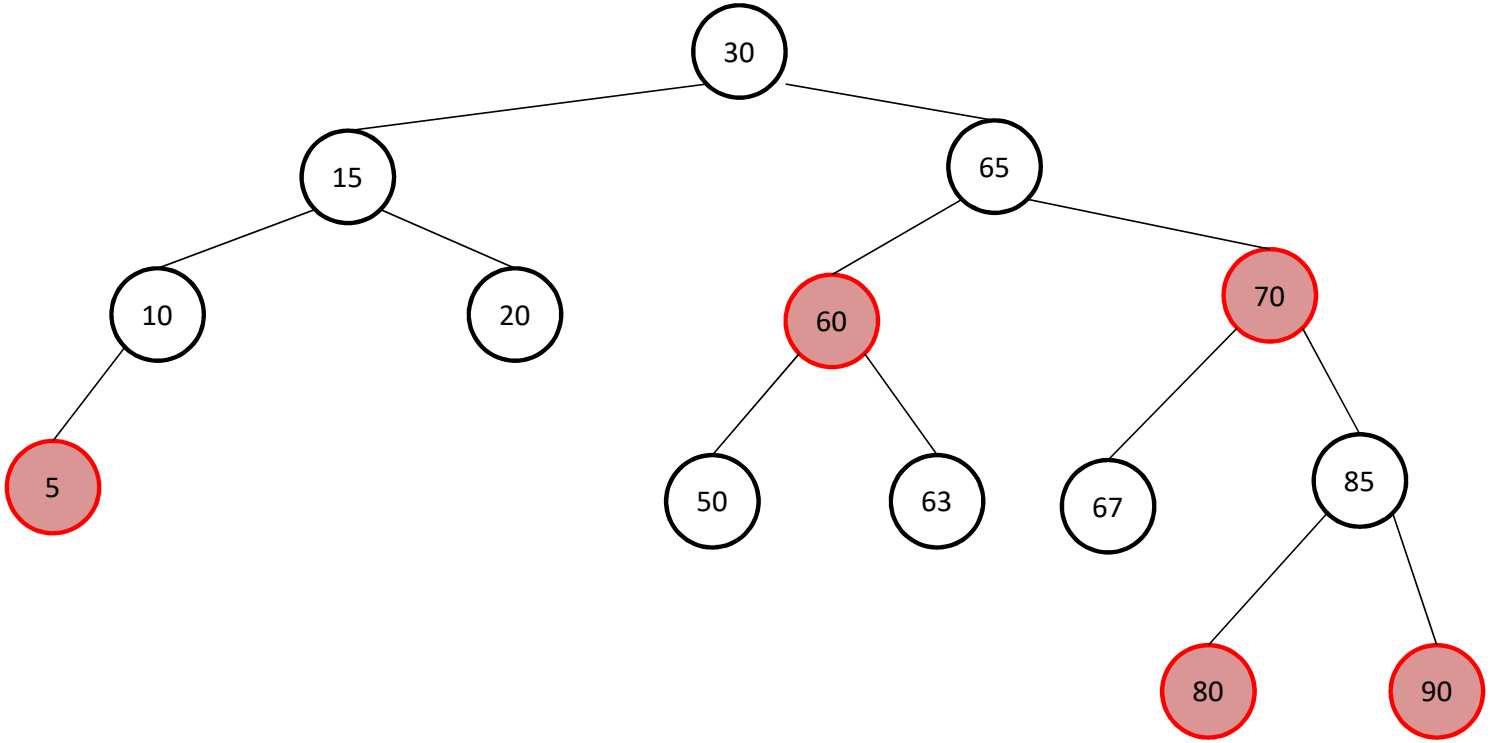
Double rotation (rotation 1)

Exercise: Insert 64 into this tree



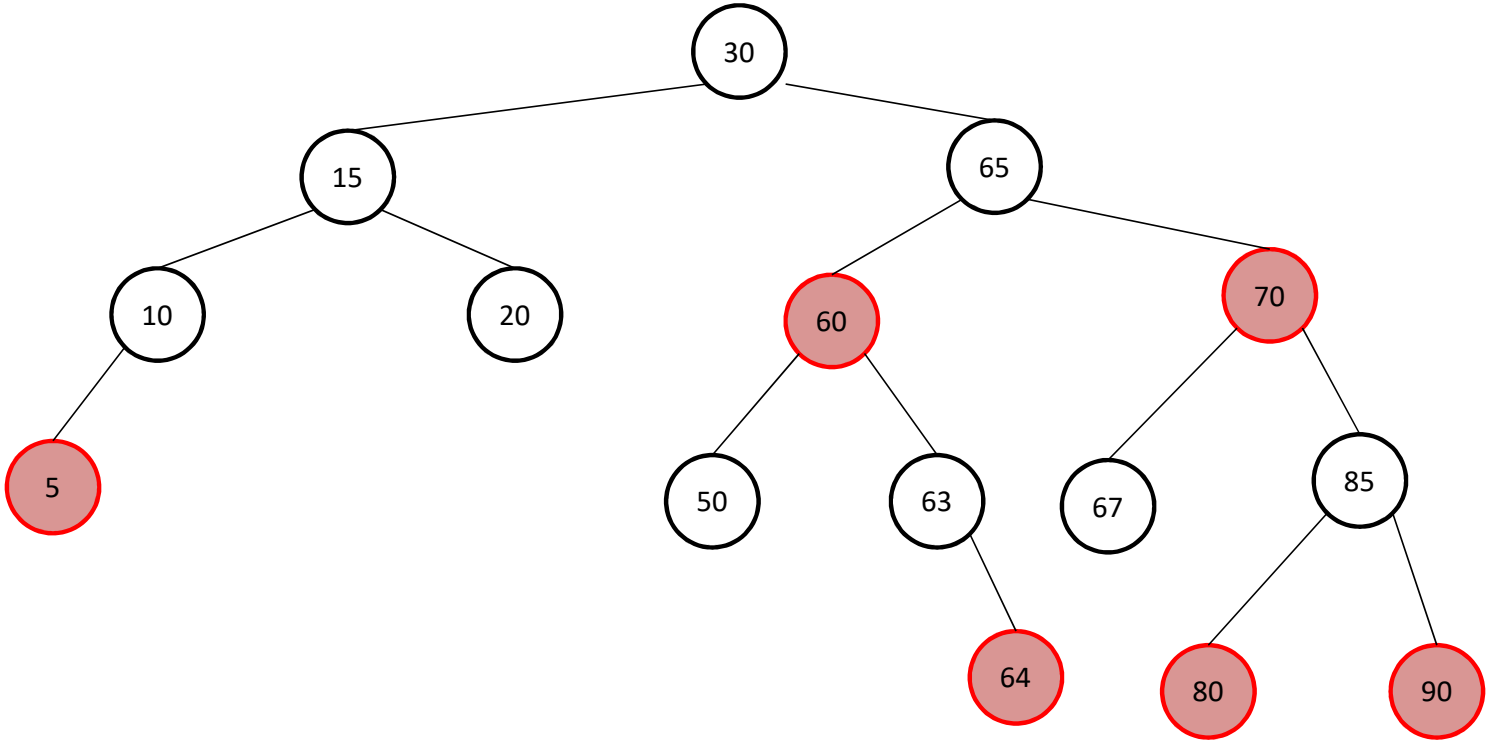
Double rotation (rotation 2)

Exercise: Insert 64 into this tree



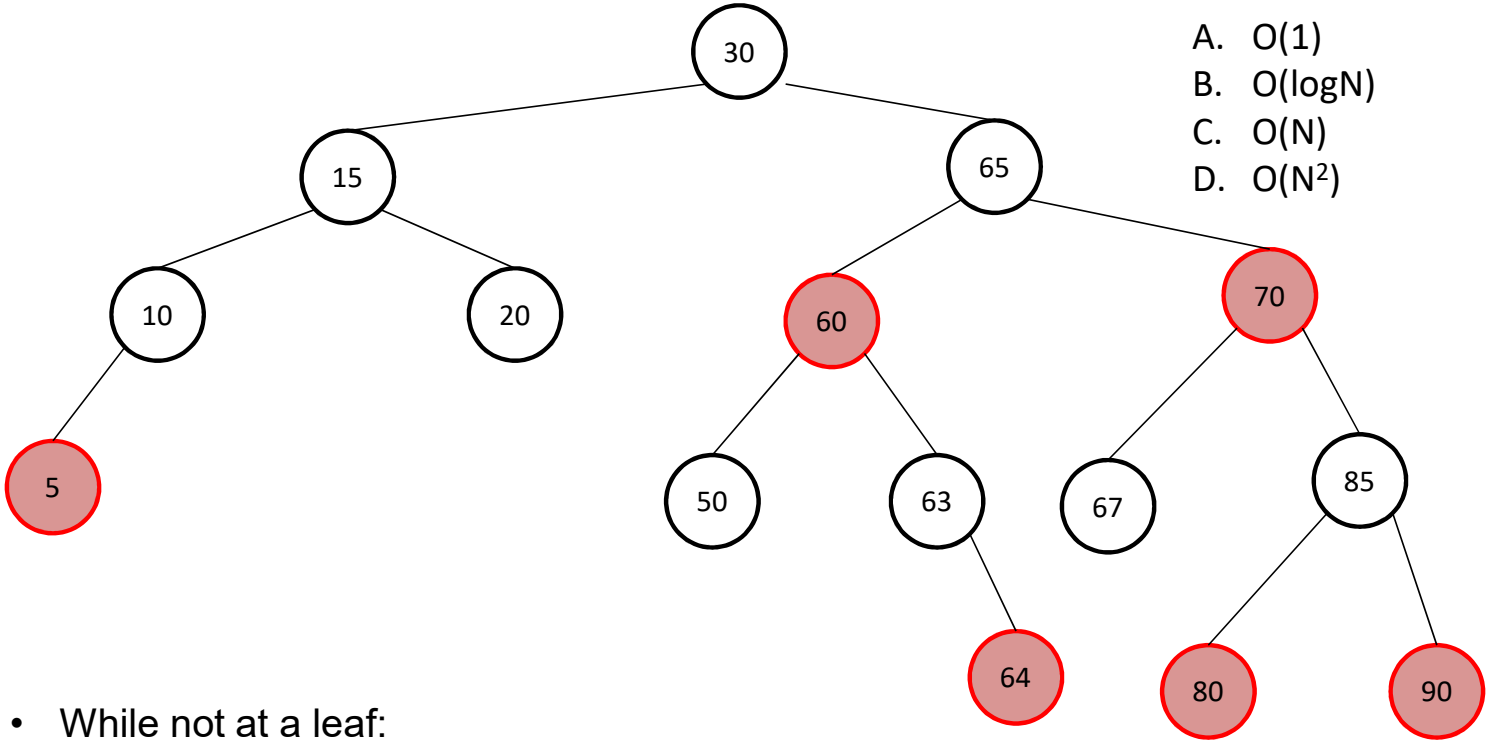
Recolor

Exercise: Insert 64 into this tree



Insert

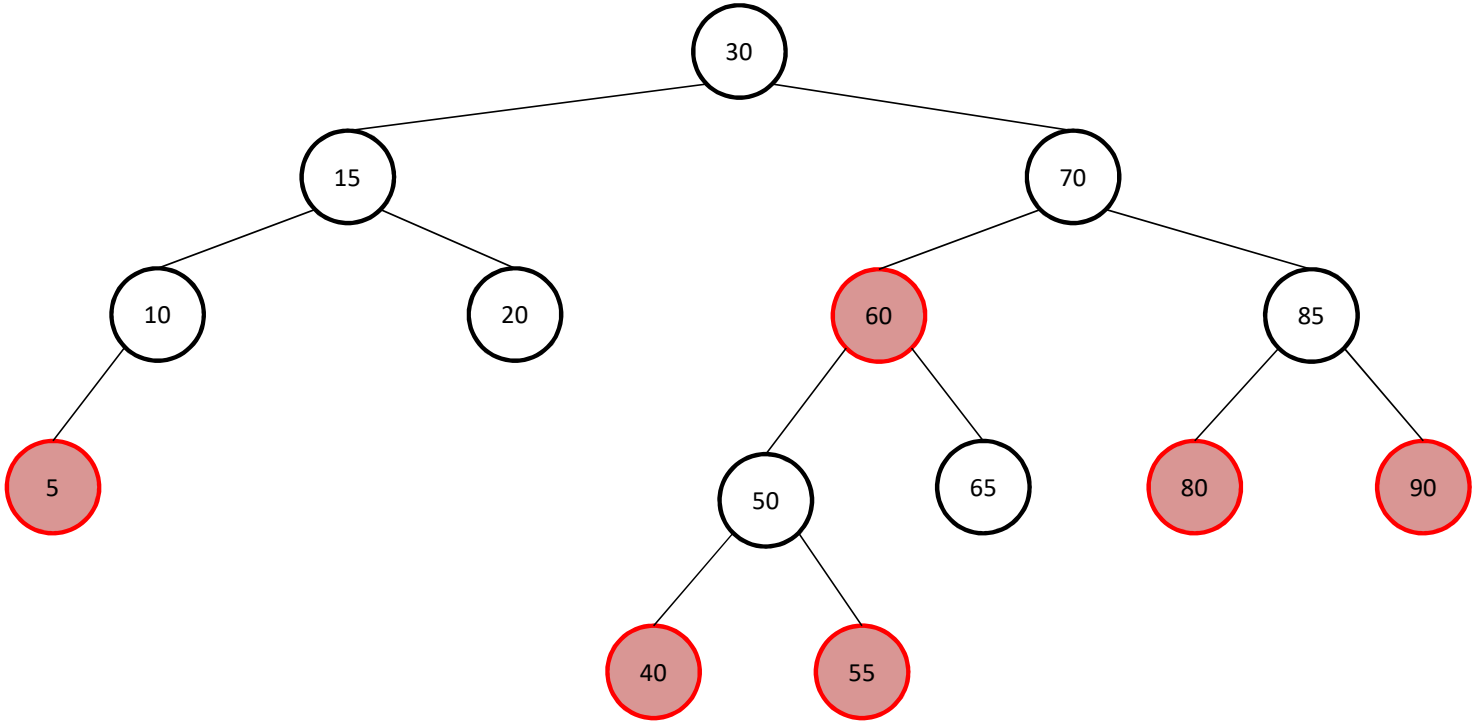
What is the Big-O running time of insert?



A. O(1)
B. O(logN)
C. O(N)
D. $O(N^2)$

- While not at a leaf:
  - Move down the tree to where node should be placed
  - If you encounter a node with two red children, recolor, then perform any necessary rotations to fix the tree
- Insert the node
- Perform any necessary rotations to fix the tree

Red-Black Trees vs. AVL trees

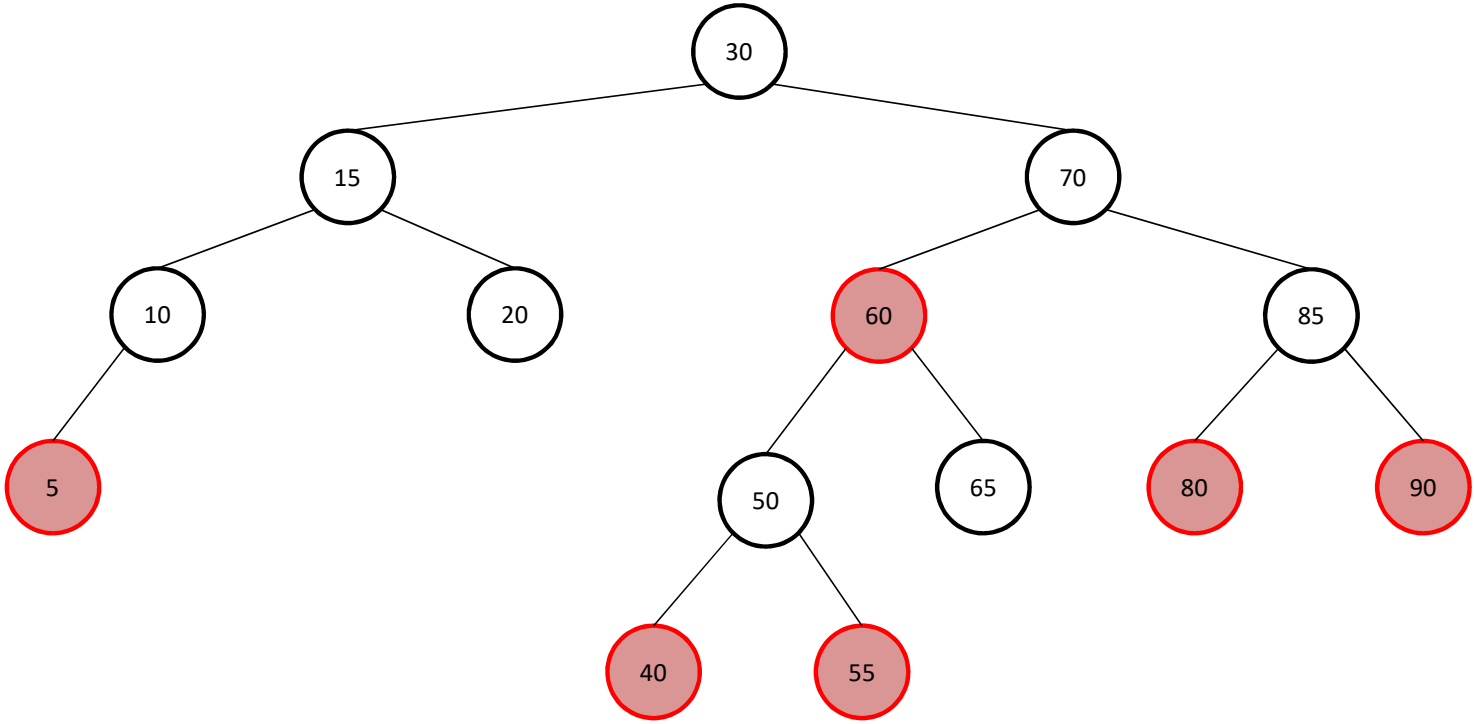


Is this an AVL tree?

A. Yes

B. No

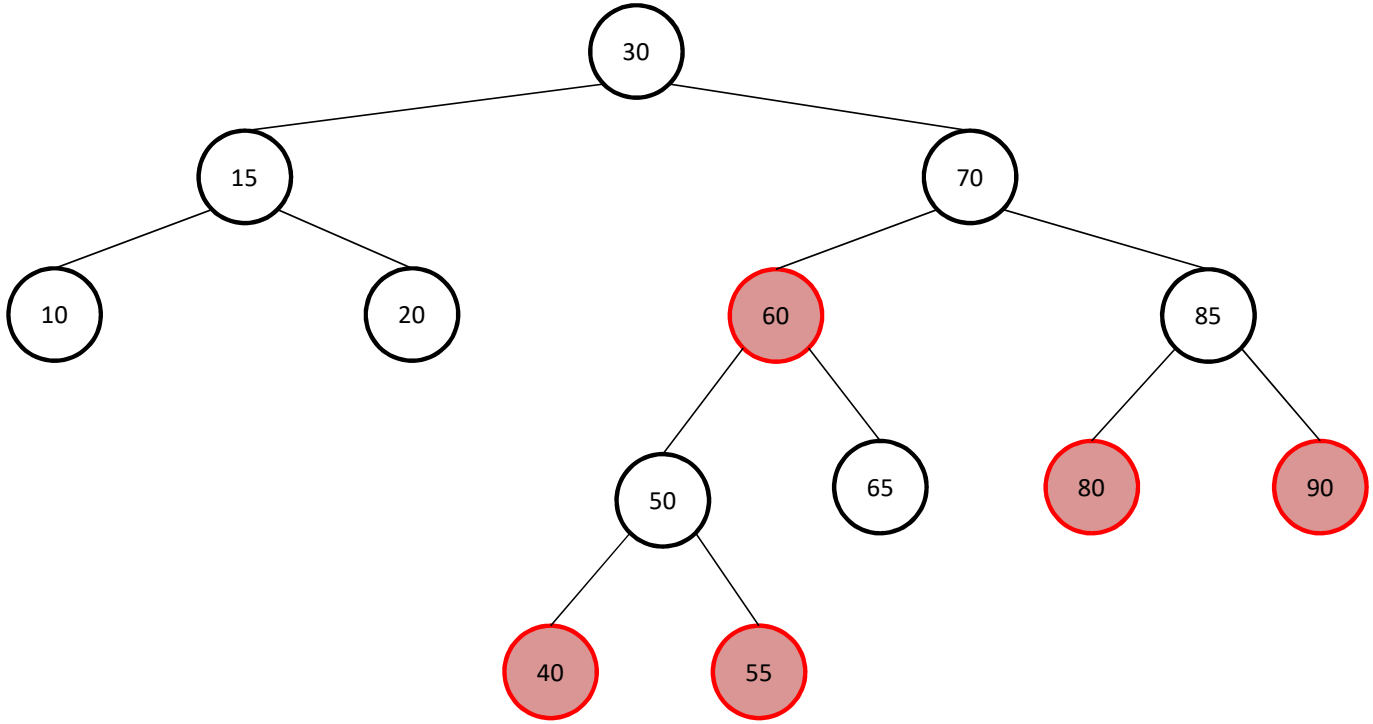
Red-Black Trees vs. AVL trees



Is this an AVL tree?
Yes

Are all red black trees AVL trees?
A. Yes
B. No

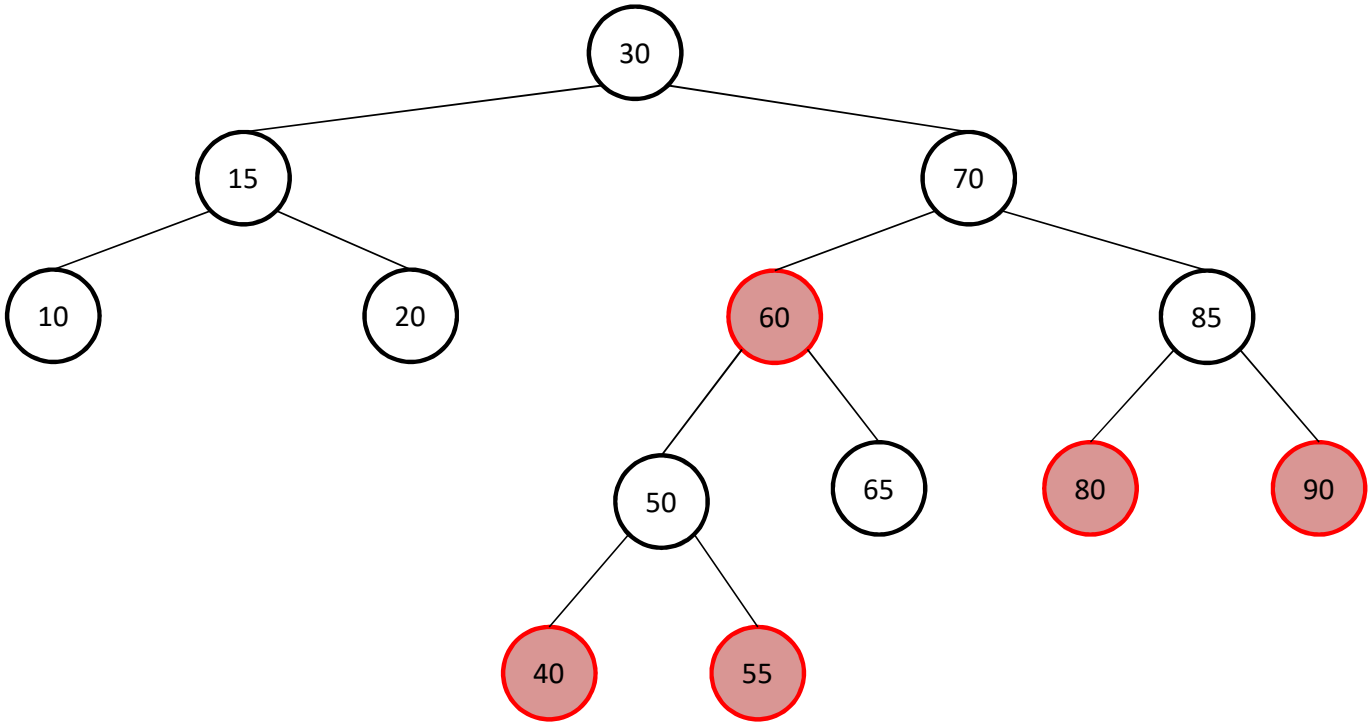
Red-Black Trees



Is this an AVL tree?
(Not anymore)

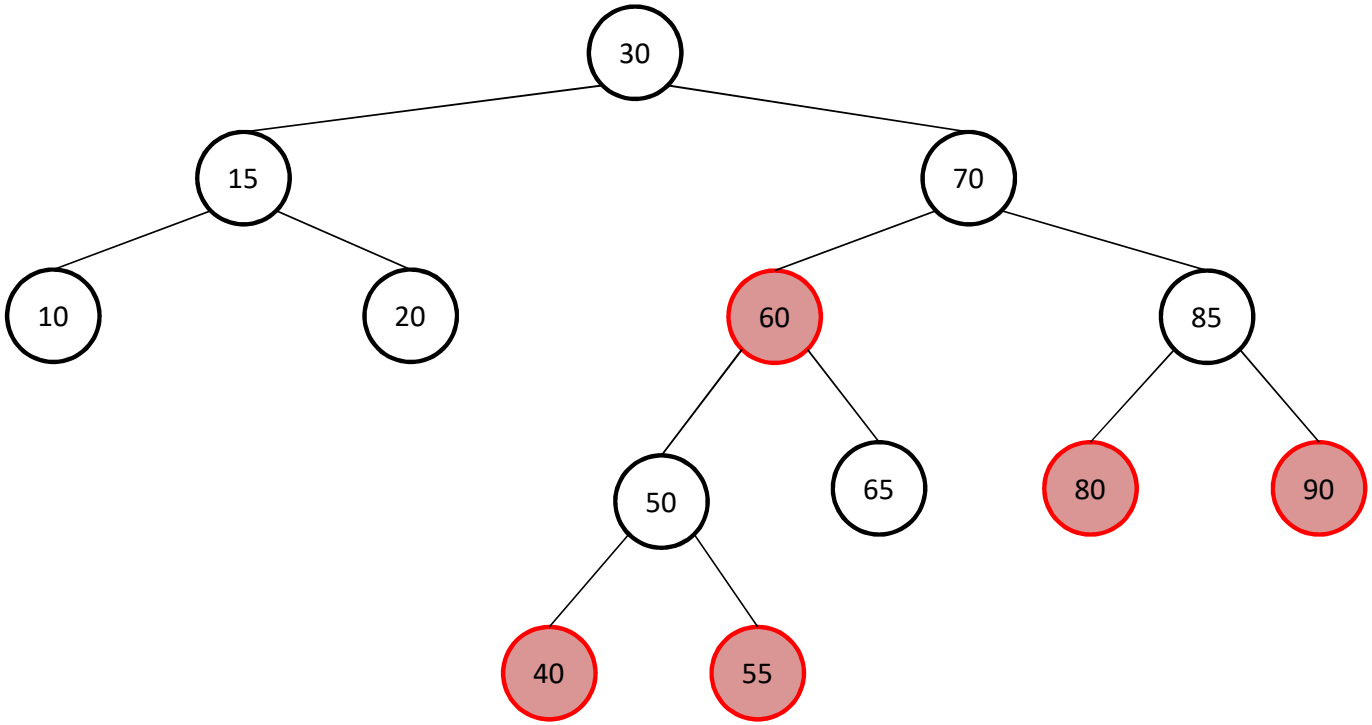
Are all red black trees AVL trees?
No

Why use Red-Black Trees



Fast to insert, slightly longer to find (but still guaranteed O(log(N)))

Why use Red-Black Trees



Faster to insert (than AVL): RBT insertion traverses the tree once instead of twice
Slower to find (that AVL): RBTs are generally slightly taller than AVL trees

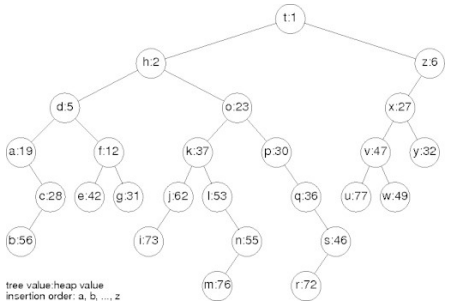
Goals for Today

- Apply Huffman's algorithm to build coding trees
- Explain how heaps work
- Use the C++ priority queue class

PA3 (Data Compression)…

- Data compression: Represent digital media using the fewest number of bits!

We will do this using trees!



11001110	01110000	01100111	00000011	11110000	11111111	01010001	11100011
00000101	11101111	00001000	01001011	11011110	00010000	10010111	10111100
00100001	00101111	01111000	01000010	01011110	11110000	01000110	01111100
00001111	10110111	00000011	01111110	10100001	00011100	01100100	11001100
00011011	11001111	11111110	11100011	11000001	00001011	11011011	01011101
11001010	11101111	10010010	10010101	11011111	00100100	00101001	00011010
01010011	00110000	01010100	01010011	11101100	01100111	10111010	01011001
01000011	11101111	00010111	11010111	01001011	10111010	00011111	11110000
10001100	10010110	00001010	00010000	10011000	00111011	00100000	11011100
00011010	01101101	00000011	11001100	11001100	11001110	01111001	01101101
10010100	10001101	01100001	01011010	01010010	10111011	10010101	11011100

Text, video, audio

All data is bits!

Fixed length encoding

- Fixed length: each symbol is represented using a fixed number of bits
- For example for the symbols ‘s’, ‘p’, ’a’, ‘m’ one possible encoding is:

spamspamspam
spamspamspam

Text file

000110110001101100011011
000110110001101100011011
000110110001101100011011

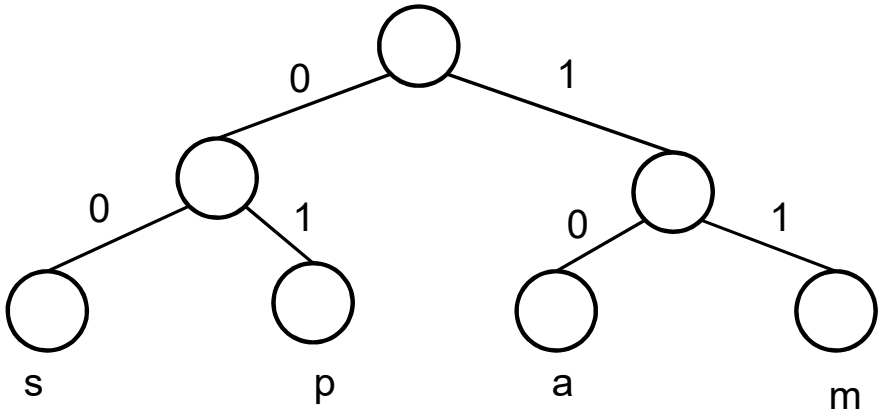
Encoded file

Symbol	Code word
s	00
p	01
a	10
m	11

For a dictionary consisting of M symbols, what is the minimum number of bits needed to encode each symbol (assume fixed length binary codes) ?

A. 2^M B. M C. M/2 D. ceil($\log_2$ M) E. None of these

Binary codes as Binary Trees



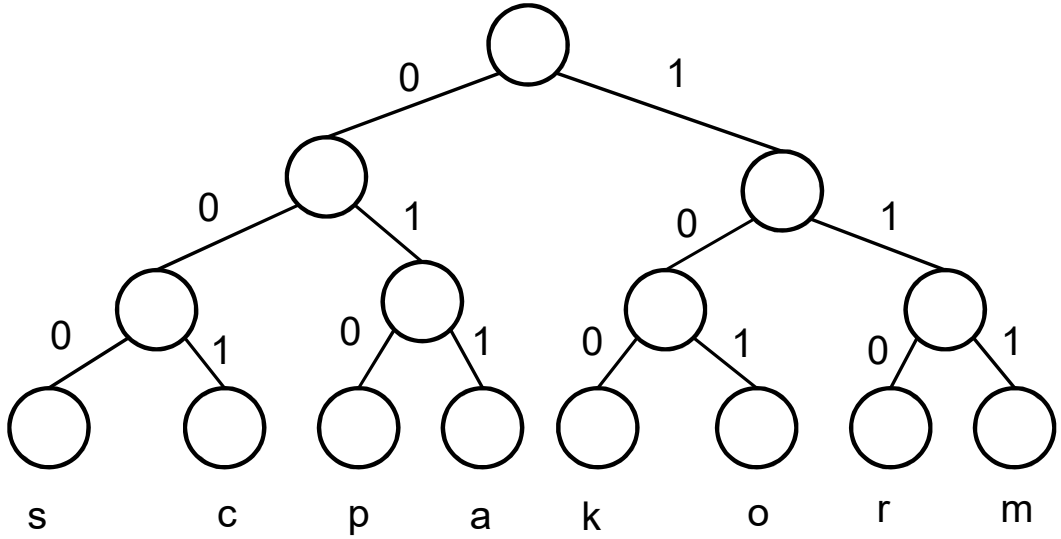
Code A

Symbol	Codeword
s	00
p	01
a	10
m	11

- Do we need to be constrained to fixed length encoding?
- What if certain symbols appeared more often than others?

- Symbols are leaf nodes
- Root to leaf node gives the codeword for each symbol
- Once we have the tree we can encode and decode data
- Given the tree
  - Encode the string ‘papa’
  - Decode the binary sequence ‘01101100’

Decoding on binary trees, another example



Decode the bitstream 110101001100 using the given binary tree

A. scam

B. rork

C. rock

D. korp

- Do we need to be constrained to fixed length encoding?
- What if certain symbols appeared more often than others?

Variable length codes

ssssssssssssssss
ssppppaampamm

Text file

Symbol	Counts
s	18
p	6
a	3
m	3

Symbol	Frequency
s	0.6
p	0.2
a	0.1
m	0.1

Code A

Symbol	Codeword
s	00
p	01
a	10
m	11

Code B

Symbol	Codeword
s	0
p	1
a	10
m	11

Average length (code A) = 2 bits/symbol

Average length (code B) = 0.6 *1 +0.2 *1 + 0.1* 2+ 0.1*2
= 1.2 bits/symbol

Comparing encoding schemes

sssssssssssssss
ssppppaampamm

Text file

Symbol	Counts
s	18
p	6
a	3
m	3

Symbol	Frequency
s	0.6
p	0.2
a	0.1
m	0.1

Code A

Symbol	Codeword
s	00
p	01
a	10
m	11

Code B

Symbol	Codeword
s	0
p	1
a	10
m	11

Is code B better than code A?

A. Yes
B. No
C. Depends

Variable length codes

Variable length codes have to necessarily be prefix codes for correct decoding

A *prefix code* is one where no symbol's codeword is a prefix of another

Code A

Symbol	Codeword
s	00
p	01
a	10
m	11

Code B

Symbol	Codeword
s	0
p	1
a	10
m	11

Code B is not a prefix code

Use Huffman's algorithm to produce the minimal average-length code!

ssssssssssssssss
ssppppaampamm

Text file

Symbol	Counts
s	18
p	6
a	3
m	3

Symbol	Frequency
s	0.6
p	0.2
a	0.1
m	0.1

Code A

Symbol	Codeword
s	00
p	01
a	10
m	11

Your turn: Apply Huffman's algorithm to the following symbols with the given frequencies

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

PA3: encoding/decoding

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 code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:

1. Read the file header (which contains the code) to recreate the tree

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

PA3: encoding/decoding

ENCODING:

1.Scan text file to compute frequencies

2.Build Huffman Tree

3.Find code for every symbol (letter)

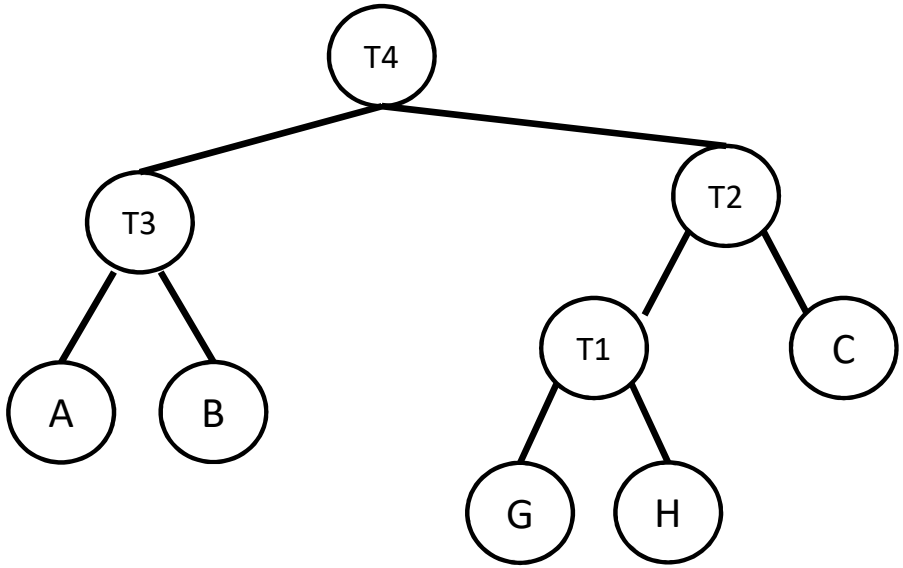
4.Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:

1. Read the file header (which contains the code) to recreate the tree

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

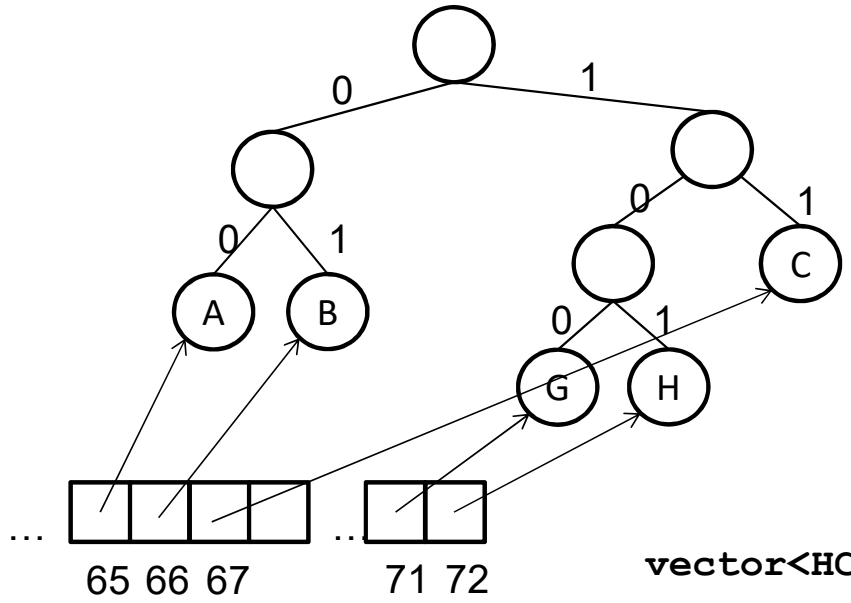
Encoding a symbol: let's think implementation!



- 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?

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)

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

PA3: encoding/decoding

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 code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:

1. Read the file header (which contains the code) to recreate the tree

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

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.

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:

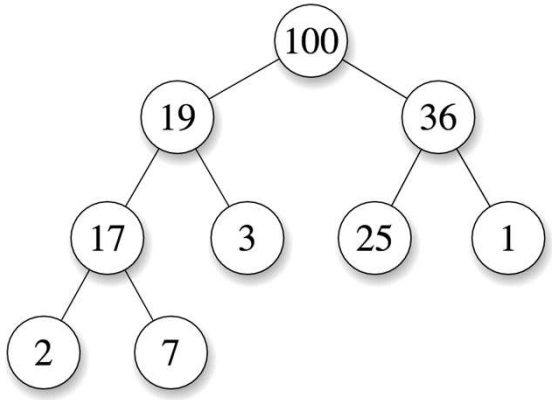
A. A list
B. A BST
C. A priority queue (heap)

Aside: Heaps

Have you seen a heap? A. Yes B. No C. Yes, but I don't remember them

Aside: Heaps

Have you seen a heap? A. Yes B. No C. Yes, but I don't remember them



if P is a parent node of C, then the *key* (the *value*) of P is either greater than or equal to (*in a max heap*) or less than or equal to (*in a min heap*) the key of C. The node at the "top" of the heap (with no parents) is called the *root* node

https://en.wikipedia.org/wiki/Heap_(data_structure)

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;
```

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` and `b` will come out before `a`**

Priority Queues in C++

```
#ifndef HCNODE_H
#define HCNODE_H
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
```

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
```

Is this implementation of operator< correct to use with the C++ priority queue (which uses a MAX-heap)?

A. Yes
B. No

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);
```

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. Adds a copy of the pointer to the node into the priority queue

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

Using std::priority_queue in Huffman's algorithm

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?

- 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 want to tell the priority_queue to first dereference the HCNode pointers it contains, and then apply operator<
- How to do that? We need to provide the priority_queue with a Compare class

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() that performs the required comparison:

comp(a, b) returns True if priority of a < priority of b (hence, ‘b’ will be ahead of ‘a’ in Queue)

```
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.

PA3: encoding/decoding

ENCODING:

1.Scan text file to compute frequencies (Monday will help)

2.Build Huffman Tree

3.Find code for every symbol (letter)

4.Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file (Monday will help)