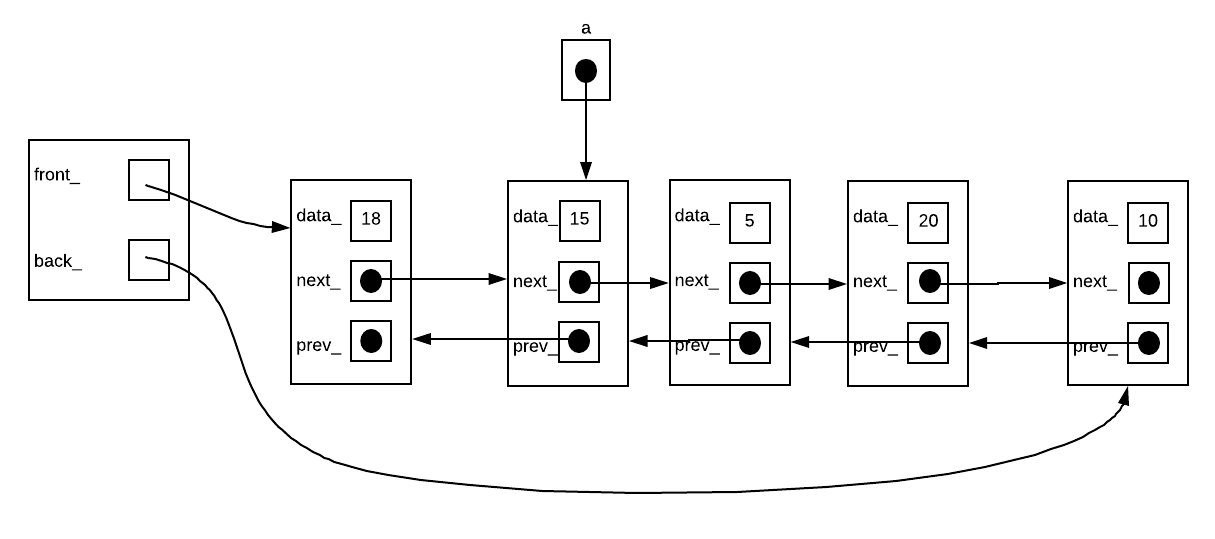
**BTP500 – Final Exam Review**

1. Below is a linked list



What code would you have to write to move node (15) one place to the right – between (5) and (20). Use one intermediary node if necessary:

Answer:

Node \*b = a->next\_;

a->prev\_->next\_ = a->next\_;

a->next\_->prev\_ = a->prev\_;

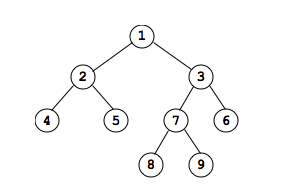
a->next\_ = b->next\_;

a->prev\_ = b;

b->next\_->prev\_ = a;

b->next\_ = a;

1. For the following tree:



1. Show the root node
2. Show the leaf nodes
3. Show the sibling of node 5
4. Show the uncle of node 8
5. Show the grandparent of node 9
6. Show the ancestors of node 4
7. Show the descendants or successors of node 3
8. What is the depth of node 7?
9. What is the height of node 7?
10. What is the height of the tree?
11. Show the preorder traversal of this tree
12. Show the inorder traversal of this tree
13. Show the postorder traversal of this tree
14. Show the breadth-first traversal of this tree
15. Is the tree complete?
16. Is the tree full?
17. Is the tree perfectly balanced?
18. Is the tree height balanced?

Answers: See notes…

1. AVL Trees, to practice inserts see:

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

1. Red-Black Trees, to practice inserts see:

<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

1. 2-3 Trees, to practice inserts see:

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

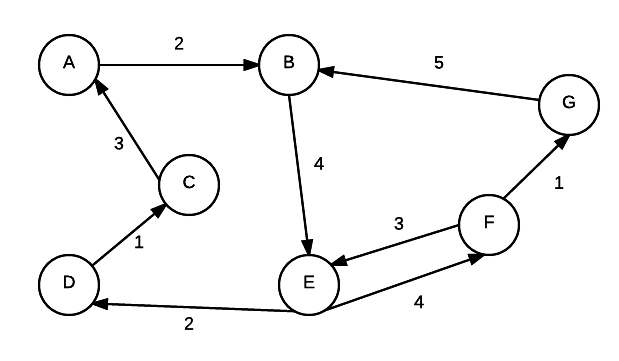
1. The Heap, to practice inserts and removal see:

<https://www.cs.usfca.edu/~galles/visualization/Heap.html>

1. What is P vs NP? See your Lab6, your notes or see:

<https://www.youtube.com/watch?v=YX40hbAHx3s&t=551s>

1. Draw the adjacency matrix and adjacency list for (see notes for answer):



1. What do the following terms in relation to graphs mean (see notes for answer)?
   1. Adjacent nodes
   2. Edge weight/cost
   3. Path
   4. Simplepath
   5. Pathlength
   6. Cycle
   7. Strongly connected
   8. Weakly connected
2. For a Binary tree as shown:

class BST{

struct Node{

int data\_;

Node\* left\_;

Node\* right\_;

Node(int dat){

data\_=dat;

left\_=right\_=nullptr;

}

};

Node\* root\_;

...

};

Write the following functions:

1. Print Preorder Traversal
2. Print Inorder Traversal
3. Print PostOrder Traversal
4. Find the Maximum node.
5. Find the Minimum node.
6. Insert a node.
7. Find a node.
8. Find the height of a tree (useful in determining whether the tree is balanced).
9. Print Breadth-First Traversal (by default left to right, could also be right to left).

Answers: See BinaryTree.cpp of Week9 and see the next few pages

void PrintPreorder(Node \*node)

{

if (node == NULL)

{

return;

}

cout << " data:" << node->data\_ << endl;

PrintPreorder(node->left\_);

PrintPreorder(node->right\_);

}

void PrintInorder(Node \*node)

{

if (node == NULL)

{

return;

}

PrintInorder(node->left\_);

cout << " data:" << node->data\_ << endl;

PrintInorder(node->right\_);

}

void PrintPostorder(Node \*node)

{

if (node == NULL)

{

return;

}

PrintPostorder(node->left\_);

PrintPostorder(node->right\_);

cout << " data:" << node->data\_ << endl;

}

Node\* FindMax(Node \*node)

{

if (node == NULL)

{

/\* There is no element in the tree \*/

return NULL;

}

if (node->right\_) /\* Go to the right sub tree to find the max element \*/

return FindMax(node->right\_);

else

return node;

}

Node\* FindMin(Node \*node)

{

if (node == NULL)

{

/\* There is no element in the tree \*/

return NULL;

}

if (node->left\_) /\* Go to the left sub tree to find the min element \*/

return FindMin(node->left\_);

else

return node;

}

Node \* Insert(Node \*node, int data)

{

if (node == NULL)

{

Node \*temp;

temp = (Node \*)malloc(sizeof(Node));//Should use new

temp->data\_ = data;

temp->left\_ = temp->right\_ = NULL;

return temp;

}

if (data > (node->data\_))

{

node->right\_ = Insert(node->right\_, data);

}

else if (data < (node->data\_))

{

node->left\_ = Insert(node->left\_, data);

}

/\* Else there is nothing to do as the data is already in the tree. \*/

return node;

}

Node \* Find(Node \*node, int data)

{

if (node == NULL)

{

/\* Element is not found \*/

return NULL;

}

if (data > node->data\_)

{

/\* Search in the right sub tree. \*/

return Find(node->right\_, data);

}

else if (data < node->data\_)

{

/\* Search in the left sub tree. \*/

return Find(node->left\_, data);

}

else

{

/\* Element Found \*/

return node;

}

}

int height(Node \*node)//See AVLTree.cpp of week 10

{

if (node == NULL)

/\* Element is not found \*/

return NULL; /\* tree is empty \*/

return 1 + max(height(node->left), height(node->right));

}

int max(int a, int b) { return (a >= b)? a: b;}

void Tree::breadthFirst()//See notes of week 11

{

/\* Temporary queue. \*/

Queue queue;

Tree \*traverse;

/\*

\* Gotta put something in the queue initially,

\* so that we enter the body of the loop.

\*/

queue.insert(root);

while (!queue.isEmpty()) {

traverse = queue.remove();

*Visit the node pointed to by* ***traverse****.*

/\*

\* If there is a left child, add it

\* for later processing.

\*/

if (traverse->getLeftSubTree() != NULL)

queue.insert(traverse->getLeftSubTree());

/\*

\* If there is a right child, add it

\* for later processing.

\*/

if (traverse->getRightSubTree() != NULL)

queue.insert(traverse->getRightSubTree());

}

}