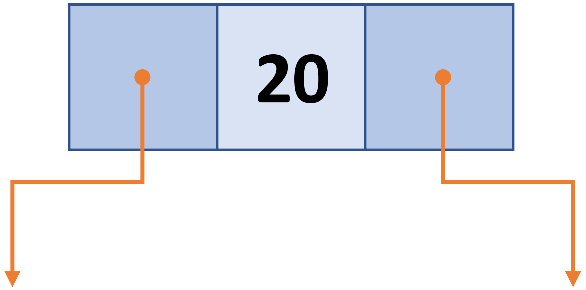
# BST with Parent Pointer

Varad Deshmukh

# BST Node Without Parent

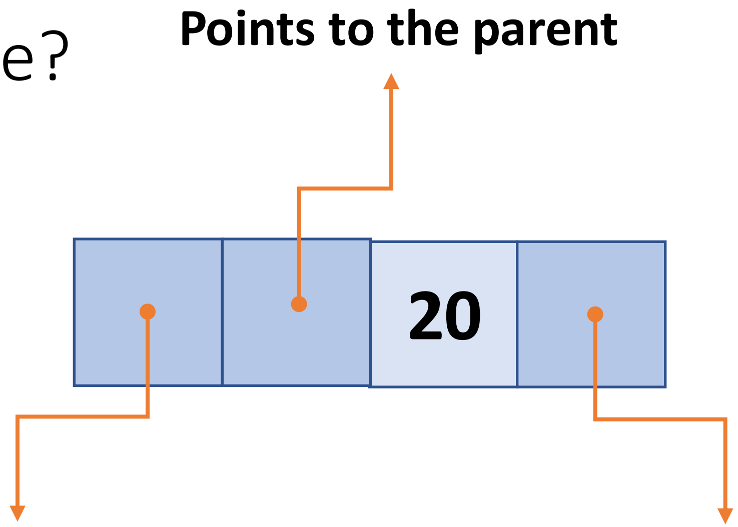
- What does each node look like?



```
struct Node {
    Node * left;
    int data;
    Node * right;
}
```

# Adding a Parent Pointer

- What does each node look like?



```
struct Node {
        Node * left;
        Node * parent;
        int data;
        Node * right;
}
```

# Why a parent pointer?

- Allows you to jump up one level easily.
- Iterative versions of the certain recursive functions such as node deletion and traversal become easier.
- Useful for node operations such as rotation when only the pointer to the node is available to the function (Assignment 7).
- Useful in special BSTs such as Red-Black Trees or AVL Trees.
- Often trades space for speed:
  - Operations become faster but extra space allocated at each node to store parent pointer.

# Insertion Without Parent

```
Node * addNode(Node * node, int key) {
   /* Base case: */
   if (node == NULL) return new Node(key);

   /* Search in the left subtree */
   if (search_key < node->value) {
         node->left = addNode(node->left, key);
   }

   /* Search in the right subtree */
   if (search_key > node->value) {
         node->right = addNode(node->right, key);
   }
   return node;
}
```

# Insertion With Parent

```
Node * addNode(Node * node, int key) {
    /* Base case: */
    if (node == NULL) return new Node(key);

    /* Search in the left subtree */
    if (search_key < node->value) {
          node->left = addNode(node->left, key);
          node->left->parent = node;
    }

    /* Search in the right subtree */
    if (search_key > node->value) {
          node->right = addNode(node->right, key);
          node->right->parent = node;
    }
    return node;
}
```

# Insertion With Parent

```
Node * addNode(Node * node, int key) {
    /* Base case: */
    if (node == NULL) return new Node(key);

    /* Search in the left subtree */
    if (search_key < node->value) {
          node->left = addNode(node->left, key);
          node->left->parent = node;
    }

    /* Search in the right subtree */
    if (search_key > node->value) {
          node->right = addNode(node->right, key);
          node->right->parent = node;
    }
    return node;
}
```

**Change the recursive traversal part only!**

Set the parent pointer of the left or right child to the current node after returning.

# Deletion Without Parent

```
Node * BST::deleteNode(Node * node, int key) {
    /* Base case: */
    if (node == NULL) return NULL;

    /* Search in the left subtree */
    else if (key < node->value) {
          node->left = deleteNode(node->left, key);

    }

    /* Search in the right subtree */
    else if (key > node->value) {
          node->right = deleteNode(node->right, key);

    }
    // We found the node with the value, now delete.
    else {
```

# Deletion With Parent

```
Node * BST::deleteNode(Node * node, int key) {
    /* Base case: */
    if (node == NULL) return NULL;

    /* Search in the left subtree */
    else if (key < node->value) {
            node->left = deleteNode(node->left, key);
            if (node->left) node->left->parent = node;
    }

    /* Search in the right subtree */
    else if (key > node->value) {
            node->right = deleteNode(node->right, key);
            if (node->right) node->right->parent = node;
    }
    // We found the node with the value, now delete.
    else {
```

**Change the recursive traversal part only!**

Set the parent pointer of the left or right child to the current node after returning.

**Deletion part in the else block stays the same.**