CSPB 2270 Exam 1 Study Guide

Binary Search Trees

Overview

Binary Search Trees (BSTs) are a type of binary tree data structure that maintain a specific order of elements. In a BST, each node has a key or value, and the keys of nodes in the left subtree are less than the key of the current node, while the keys of nodes in the right subtree are greater. This property enables efficient searching, insertion, and deletion operations. BSTs offer fast average case performance for these operations, with a time complexity of $\mathcal{O}(\log(n))$ in balanced trees. They are commonly used in scenarios that require fast searching or ordered traversal of elements, such as dictionary implementations, range queries, and efficient data organization.

A Binary Search Tree (BST) and a linked list differ in their structure and organization of data. A BST is a binary tree where nodes are ordered based on their keys, allowing for efficient search, insertion, and deletion operations. In contrast, a linked list is a linear structure where elements are connected via pointers, without a specific ordering. Traversing a linked list requires sequentially following the next pointers, while a BST enables faster search and manipulation of elements due to its ordered nature.

A balanced tree is a tree data structure in which the heights of the left and right subtrees of any node differ by at most one. It ensures that the tree is evenly balanced and maintains efficient operations. Balanced trees are desirable because they provide improved performance for various operations such as searching, inserting, and deleting elements. By keeping the tree balanced, the height remains relatively low, which reduces the time complexity of operations and ensures that the worst-case scenario is avoided. Common examples of balanced trees include AVL trees, red-black trees, and B-trees.

BST's have some common terminology that are useful to describe certain aspects of the structure. The terminology that we use to describe a BST can be seen below:

- **Children:** Children are defined as nodes that are connected to a parent node, either left or right, in the hierarchical structure of the tree
- **Complete:** A BST where if all levels, except the last level, contain all possible nodes and all nodes in the last level are as far left as possible
- **Depth:** Depth is defined as the number of edges on the path from the root to the node in question
- **Edge:** An edge is a link from a parent node to its child node
- **Full:** A BST where every node contains 0 or 2 children
- **Height:** The height of a tree is the largest depth of any node in the tree
- **Internal Node:** An internal node is a node with at least one child
- **Leaf:** A leaf is a node with no children
- **Level:** The level of a BST is defined as nodes that have the same depth
- **Parent:** A parent is a node where one of more child nodes are directly connected to it
- **Perfect:** A BST where if all internal nodes have 2 children and all leaf nodes are at the same level
- **Root:** The root of a BST is referred to as the topmost node of the tree structure (The root node has no parent node)

The terms: **Full**, **Complete**, and **Perfect** define special types of BST's. The rest of the terms in the above list are components of the BST

BST's use a technique called recursion. Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem. It is useful in dealing with trees because trees are recursive data structures. Each node in a tree can be viewed as a smaller tree itself, consisting of its children and their descendants. Recursion allows us to traverse and manipulate tree structures by breaking down the problem into smaller sub problems that can be solved recursively. It simplifies the code by abstracting the complex tree operations into simpler recursive calls, leading to concise and elegant solutions. Additionally, recursion provides a natural way to handle tree-related algorithms, such as tree traversal, searching, insertion, deletion, and various other tree-based computations.

Common BST Operations

BST's require a number of operations for the functionality of it to be optimal. A lot of these operations can also be seen in linked lists. The operations that are used in BST's are the following:

- **Contains:** The contains function in a Binary Search Tree (BST) is used to check if a given value is present in the tree.
- **Get Node:** The getnode function in a Binary Search Tree (BST) is used to search for a node with a specified value in a tree.
- **Height:** The height operation in a BST recursively calculates the maximum number of edges from the root to any leaf node, providing a measure of the tree's vertical depth.
- **Initialize:** Initializing a node in a Binary Search Tree (BST) means creating a new node with a given value and setting its left and right child pointers to null or nullptr, indicating an empty subtree.
- **Insert:** The insert function in a Binary Search Tree (BST) is responsible for adding a new node to the tree while maintaining the binary search property, where values smaller than the current node are placed in the left subtree, and values larger than the current node are placed in the right subtree.
- **In Order:** An in-order traversal function in a Binary Search Tree (BST) visits all the nodes in the tree in ascending order of their values.
- **Post Order:** A post-order traversal function in a Binary Search Tree (BST) visits all the nodes in the tree in a specific order.
- **Pre Order:** A pre-order traversal function in a Binary Search Tree (BST) visits all the nodes in the tree in a specific order.
- **Remove:** The remove function in a Binary Search Tree (BST) is used to delete a node from the tree while preserving the binary search property. It involves finding the node to be removed, handling different cases based on the number of children the node has, and reorganizing the tree as necessary to maintain its structure and properties.
- **Size:** The size function in a Binary Search Tree (BST) is used to determine the number of nodes in a tree.
- **To Vector:** The tovector function in a Binary Search Tree (BST) is used to add values in a tree to a vector.

We can now look at each of these operations individually. The first operation that we will look at is the contains operation.

Contains In BST

Contains

Below is an example of the implementation of the contains operation in a BST in the context of C++:

```
/*  Contains - This function checks to see if a specific value is present in our
*              BST
*   Input:
*     subt - A smart pointer pointing to a (bst_node) node in a BST
*     data - An integer whose value is being searched for in a BST
*   Algorithm:
*     * We first check to see if the current node in the tree is null, if it is, we
*       return false
*     * We then check if the current nodes data member "data" is equal to the input
*       parameter "data"
*       * If it is equal to it, we return true, indicating this value exists in our
*         tree
*     * If neither of the two previous conditions are met, then we search through
*       the subtrees of this BST
*       using a recursive call to "Contains"
*       * We then return the boolean value after searching through each subtree
*         * If both are false, then false is returned
*         * If atleast one of these are true, then we return true
*   Output:
*     We return a boolean value indicating whether or not a value is present in
*     a BST
*   Reference:
*     The skeleton of this algorithm was referenced from the zyBooks chapter 7.11
```

```
*/
bool BST::Contains(shared_ptr<bst_node> subt, int data){
    if (subt == nullptr) {
        return false;
    }
    else if (subt->data == data) {
        return true;
    }
    else {
        return Contains(subt->left, data) || Contains(subt->right, data);
    }
}
```

The above function takes in a node that is present in a BST and searches for a specific value in the list. It utilizes recursion to search both the left and right subtrees of the current node and returns a boolean value determining if the value is present in the tree. The time complexity of this algorithm is the following:

Best Case Time Complexity	Worst Case Time Complexity
$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$

The best case time complexity occurs when the tree is balanced. The worst case time complexity occurs when the node doesn't exist or if the node is at the leaf level of the tree.

The next operation that we will look at is the GetNode operation.

Get Node In BST

Get Node

Below is an example of the implementation of the Get Node operation in a BST in the context of C++:

```
/*  GetNode - This function returns a pointer if a node is present in a BST
*   Input:
*     subt - A smart pointer pointing to a (bst_node) in a BST
*     data - An integer whoe value is being searched for in a BST
*   Algorithm:
*     * We first check to see if the current node in the BST is null, if it is,
*       return NULL
*     * If the current node is not null, we check to see if the current nodes
*       "data" member matches the input parameter "data", if it does, we return
*       the current pointer of the node
*     * If neither of the two conditions are met, we make a recursive call to the
*       left and right subtrees of the BST
*     * We then iterate through the previous steps of the algorithm to determine
*       if a data value is present in our BST
*   Output:
*     We return a pointer to our node depending on if prior conditions are met
*     when searching for a value
*/
shared_ptr<bst_node> BST::GetNode(shared_ptr<bst_node> subt, int data){
    if (subt == nullptr) {
        return shared_ptr<bst_node>(nullptr);
    }
    else if (subt->data == data) {
        return subt;
    }
    else {
    if (GetNode(subt->left, data)) {
        return GetNode(subt->left, data);
    }
    else {
        return GetNode(subt->right, data);
        }
```

```
    }
}
```

This operation searches a BST for a specific value and returns the node if it exists. Similar to the contains operation the GetNode operation uitlizes recursion to search the left and right subtrees of the node that is fed into the function. The time complexity of this algorithm is the following:

Best Case Time Complexity	**Worst Case Time Complexity**
$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$

The time complexity of this operation is the same as that of the operation that just searches for a value in a BST. When searching for the maximum or minimum value in a tree, the time complexity is the same for searching for a value in the tree. These time complexities are contingent upon the balancing of the tree.

The next operation that we will look at is the Height operation.

Height In BST

Height

Below is an example of the implementation of calculating the height of a BST in the context of C++:

```
int height(Node* root) {
    if (root == nullptr) {
        return 0;  // An empty tree has height 0
    }
    else {
        // Recursively calculate the height of the left and right subtrees
        int leftHeight = height(root->left);
        int rightHeight = height(root->right);

        // Return the maximum height between the left and right subtrees, plus 1 for the
        return std::max(leftHeight, rightHeight) + 1;
    }
}
```

The above function calculates the max number of edges to the leaf level in a BST. This function utilizes the concept of recursion to find which subtree has the most edges in it. This in turn is the height of the BST.

The next operation that we will look at is the Initialize operation.

Initializing In BST

Initializing

Below is an example of the implementation of initializing a node in a BST in the context of C++:

```
/*  InitNode - This function creates a node of a BST
*   Input:
*     data - Integer value that the data member of bst_node "data" is to be set to
*   Algorithm:
*     * Create a new smart pointer named "ret" to represent a new BST node
*       (bst_node)
*     * Assign the input parameter "data" to the data member "data" of "ret" using
*       the member access operator
*     * Initialize the left and right child pointers of "ret" to a null value with
*       "nullptr"
*   Output:
*     ret - A smart pointer pointing to the newly created BST node (bst_node)
*/
shared_ptr<bst_node> BST::InitNode(int data){
    shared_ptr<bst_node> ret(new bst_node());
    ret->data = data;
    ret->left = nullptr;
```

```
    ret->right = nullptr;
    return ret;
}
```

The above function initializes a node for a BST. It sets the value of the 'data' member to the input parameter that is passed in the function. This operation has a constant time complexity of $\mathcal{O}(1)$.

The next operation that we will look at is the insert operation.

Inserting In BST

Inserting

Below is an example of the implementation of inserting into a BST in the context of C++:

```
/*  Insert - This function inserts a node into a BST
*   Input:
*     new_node - A smart pointer that represents a node in a BST
*   Algorithm:
*     * First, check if the root of our BST is null
*       * If it is, then assign the smart pointer "new_node" to the root of our BST
*         using SetRoot()
*       * If it is not, then proceed to the next step
*     * We then create a dummy smart pointer "current_node" and assign it to the
*       root of our tree
*     * The following while loop will run until the current node is null
*     * We then check what the value of our input parameter "new_node" is in
*       comparison to our current node of the BST
*     * If "new_node"'s value is less than that of the current node in our BST, we
*       do the following:
*       * We check if the left child of the current node is null
*         * If it is, then the left child of "current_node" is assigned to our
*           "new_node" that is passed as an input parameter "current_node" is then
*           assigned to null to exit the above while loop
*         * If it is not, then the "current_node" is set to the left child of
*           "current_node"
*     * If "new_node"'s value is greater than or equal to that of the current node
*       in our BST, we do the following:
*       * We check if the right child of "current_node" is null
*         * If it is, then the right child of "current_node" is assigned to our
*           "new_node" that is passed as an input parameter "current_node" is then
*           assigned to null to exit the above while loop
*         * If it is not, then the "current_node" is set to the right child of
*           "current_node"
*   Output:
*     There is no output for this function
*   Reference:
*     The skeleton of this algorithm was referenced from the zyBooks Chapter 7.6
*/
void BST::Insert(shared_ptr<bst_node> new_node){
    if (GetRoot() == nullptr) {
        SetRoot(new_node);
    }
    else {
        shared_ptr<bst_node> current_node = GetRoot();
        while (current_node != nullptr) {
            if (new_node->data < current_node->data) {
                if (current_node->left == nullptr) {
                    current_node->left = new_node;
                    current_node = nullptr;
                }
                else {
                    current_node = current_node->left;
```

```
                }
            }
            else {
                if (current_node->right == nullptr) {
                    current_node->right = new_node;
                    current_node = nullptr;
                }
                else {
                    current_node = current_node->right;
                }
            }
        }
    }
}
```

The insert operation inserts a node into a BST in the correct spot in the tree. This function operates by searching through a BST for where the node should be inserted. Similar to that of the search operation, the insert operation as the following time complexities:

Best Case Time Complexity	Worst Case Time Complexity
$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$

The time complexities of this operation are again contingent upon the balancing of the tree. This operation effecitvely searches through the BST to try and find the appropriate spot for where it should be inserted. The best case scenario for inserting into a BST occurs when it is balanced and the worst case happens when the insertion happens at the leaf level.

The next operation that we will look at is the in order traversal of a BST.

In Order Traversal In BST

In Order

Below is an example of the implementation of the in-order traversal of a BST in the context of C++:

```
/*  InOrder - This function traverses a binary search tree in order
*   Input:
*     subt - A smart pointer pointing to a node in BST
*   Algorithm:
*     * We first check to see if the node that we are looking at is null, if it is,
*       then we return that node
*     * If the node that we are examining is not null, then we proceed to checking
*       each sub tree
*       * If the left child of "subt" is not null, then we recursively call
*         `InOrder`on the left child to find the leftmost node in the left subtree,
*         the above process will repeat until a null pointer is found, indicating
*         we have reached the leftmost node of this subtree
*       * If the left child of "subt" is null, we move to the right child and
*         recursively call `InOrder` on the right child to find the leftmost node
*         in the right subtree, the above process will repeat until a null pointer
*         is found, indicating we have reached the leftmost node of this subtree
*   Output:
*     This function returns a smart pointer to the leftmost node in our BST
*   Reference:
*     The skeleton of this algorithm was referenced from the zyBooks chapter 7.8
*/
shared_ptr<bst_node> BST::InOrder(shared_ptr<bst_node> subt) {
    if (subt == nullptr) {
        return subt;
    }
    else {
        if (InOrder(subt->left)) {
            return InOrder(subt->left);
```

```
        }
        else {
            return InOrder(subt->right);
        }
    }
}
```

This function traverses a BST in order, meaning in ascending order. This function utilizes recursion to traverse the tree. It traverses in ascending order by traversing through all of the left subtrees and then the right subtrees. The time complexity of the in order traversal is $\mathcal{O}(n)$ regardless of the balancing of the tree.

The next operation that we will look at is the post order traversal of a BST.

Post Order Traversal In BST

Post Order

Below is an example of the implementation of post order traversal in a BST in the context of C++:

```
void postOrderTraversal(shared_ptr<bst_node> root) {
    if (root == nullptr) {
        return;
    }

    // Visit the left subtree
    postOrderTraversal(root->left);

    // Visit the right subtree
    postOrderTraversal(root->right);

    // Process the root node
    cout << root->data << " ";
}
```

The above function traverses a BST in post order fashion. In post-order traversal, the nodes of the BST are visited in the following order: left subtree, right subtree, and then the root. The recursive function is called first for the left subtree, then for the right subtree, and finally the root node is processed. Similar to the in order traversal, the time complexity of this operation is $\mathcal{O}(n)$ regardless of the balancing of the tree.

The next operation that we will look at is the pre order traversal of a BST.

Pre Order Traversal In BST

Pre Order

Below is an example of the implementation of pre order traversal in a BST in the context of C++:

```
void PreOrderTraversal(std::shared_ptr<bst_node> root) {
    if (root == nullptr)
        return;

    // Process current node (root)
    std::cout << root->data << " ";

    // Recursively traverse left subtree
    PreOrderTraversal(root->left);

    // Recursively traverse right subtree
    PreOrderTraversal(root->right);
}
```

The above function traverse a BST in a pre order fashion. In pre-order traversal, we visit the current node (root) first, then recursively traverse the left subtree, and finally recursively traverse the right subtree. Similar to that of the past two traversals of the tree, the time complexity of this operation is $\mathcal{O}(n)$ regardless of the balancing of the tree.

The next operation that we will look at is the remove operation of a BST.

Remove In BST

Remove

Below is an example of the implementation of the remove operation in a BST in the context of C++:

```
/*  Remove - This function removes a node in a BST for a given integer value "data"
*   Input:
*     data - Integer value that is to be searched for and removed from our BST
*   Algorithm:
*     * We first begin by creating three separate nodes (bst_node) and initialize
*       them to specific values
*       * current_node - This node is set to the root of our BST
*       * parent_node - This node is the parent node of our current node in our BST
*       * successor_node - This node is the succeeding node of our current node
*     * We then begin searching our BST by first looking at the root of our BST
*     * The first condition that is checked for is if the current node matches the
*       input parameter "data"
*       * We then have four different checks to make according to the child nodes
*         of the current node
*       * Remove Leaf Node - This occurs when the children of the current node are
*         both null
*         * We then have three conditions to check for inside of this current
*           condition
*           * If the parent node is null, then we set the root node to be null
*             This is the condition where the node that is being removed is the
*             current root of the tree
*           * If the left child of the parent node is the current node, then we
*             set the left child of the parent node to be null, we are removing
*             "current_node" from the tree
*           * Otherwise, if the right child of the parent node is the current node,
*             then we set the right child of the parent node to be null, we are
*             removing "current_node" from the tree
*       * Remove Left Child - This occurs when the right child of the current node
*         is null
*         * We then have three conditions to check for inside of this current
*           condition
*           * If the parent node is null, then we set the root node to be the left
*             child of the current node In this scenario, the current node is the
*             root of the tree and the only remaining node is the left child of the
*             current node, this is why it is set to the root of the tree
*           * If the left child of the parent node is the current node, then we set
*             the left child of the parent node to be the left child of the current
*             node, this replaces the current node with its left child after being
*             removed
*           * If the right child of the parent node is the current node, then we set
*             the right child of the parent node to be the left child of the current
*             node, this replaces the current node with its left child after being
*             removed
*       * Remove Right Child - This occurs when the left child of the current node
*         is null
*         * We then have three conditions to check for inside of this current
*           condition
*           * If the parent node is null, then we set the root node to be the right
*             child of the current node In this scenario, the current node is the
*             root of the tree and the only remaining node is the right child of
*             the current node, this is why it is set to the root of the tree
*           * If the left child of the parent node is the current node, then we set
*             the left child of the parent node to be the right child of the current
*             node, this replaces the current node with its right child after being
*             removed
```

```
*           * If the right child of the parent node is the current node, then we
*             set the right child of the parent node to be the right child of the
*             current node, this replaces the current node with its right child
*             after being removed
*       * Remove Node With Two Children - This occurs when the node to be removed
*         has two children attached to it
*         * In this scenario, we set the successor node to be the right child of the
*           current node
*         * We then set the successor node to be its left child until its left child
*           is null, effectively searching the leftmost sub tree until the
*           appropriate successor is found
*         * We then create a copy of this value, and assign it to the integer value
*           "SuccessorData"
*         * Then, a recursive call of "Remove" is made on the "successor_node"'s
*           "data" member, this is done so that we can remove the successor of the
*           current node and put it in the place of the current node
*         * The current node's "data" member is then assigned to the integer value
*           "SuccessorData", effectively placing the successor node in the position
*           of the current node
*     * If the key is not found, then we check if the current nodes data member
*       "data" is less than that of the input parameter "data"
*       * If this is the case, then the parent node is set to that of the current
*         node and the current node is set to its right child We are effectively
*         searching the right sub tree of our BST
*     * If the key is not found, then we check if the current nodes data member
*       "data" is greater than that of the input parameter "data"
*       * If this is the case, then the parent node is set to that of the current
*         node and the current node is set to its left child We are effectively
*         searching the left sub tree of our BST
*     * If neither of the above conditions are met, then that means that the value
*       being searched for does not exist in our BST
*   Output:
*     This function does not return a value
*   Reference:
*     The sekelton of this algorithm was referenced from the zyBooks chapter 7.7
*/
void BST::Remove(int data) {
shared_ptr<bst_node> current_node = GetRoot();
shared_ptr<bst_node> parent_node = nullptr;
shared_ptr<bst_node> successor_node = nullptr;
while (current_node != nullptr) {
    // Node found
    if (current_node->data == data) {
        // Remove a leaf node
        if (current_node->left == nullptr && current_node->right == nullptr) {
            if (parent_node == nullptr) {
                SetRoot(nullptr);
            }
            else if (parent_node->left == current_node) {
                parent_node->left = nullptr;
            }
            else {
                parent_node->right = nullptr;
            }
        }
        // Remove left child only
        else if (current_node->right == nullptr) {
            if (parent_node == nullptr) {
                SetRoot(current_node->left);
            }
            else if (parent_node->left == current_node) {
```

```
                parent_node->left = current_node->left;
            }
            else {
                parent_node->right = current_node->left;
            }
        }
        // Remove right child only
        else if (current_node->left == nullptr) {
            if (parent_node == nullptr) {
                SetRoot(current_node->right);
            }
            else if (parent_node->left == current_node) {
                parent_node->left = current_node->right;
            }
            else {
                parent_node->right = current_node->right;
            }
        }
        // Remove node with two children
        else {
            successor_node = current_node->right;
            while (successor_node->left != nullptr) {
                successor_node = successor_node->left;
            }
            int SuccessorData = successor_node->data;
            Remove(successor_node->data);
            current_node->data = SuccessorData;
        }
        break;
    }
        // Search to the right
        else if (current_node->data < data) {
            parent_node = current_node;
            current_node = current_node->right;
        }
        // Search to the left
        else {
            parent_node = current_node;
            current_node = current_node->left;
        }
    }
}
```

The above function removes a node from a BST and reorders the tree so that it still follows the same rules of a BST. The time complexity of this operation is the following:

Best Case Time Complexity	**Worst Case Time Complexity**
$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$

Similar to other operations, the remove operation's time complexity is contingent upon the balancing of the tree.

The next operation that we will look at is the size operation of a BST.

Size Of BST

Size Of

Below is an example of the implementation of the size operation in a BST in the context of C++:

```
/*  Size - This function counts the number of nodes that are present in a given BST
*   Input:
*     subt - A smart pointer that points to a "bst_node" object representing a BST
*   Algorithm:
```

```
*      * We first check if the input parameter "subt" is a null, if it is, then we
*        return a value of zero
*      * If the current node, "subt", is not null, then:
*        * We add 1 to our total, and the recursively call "Size" on the left and
*          right child of "subt"
*        * This process will repeat over and over until the end of the tree is
*          reached for both subtrees
*    Output:
*      This function returns an integer value representing the number of nodes in
*      a BST
*/
int BST::Size(shared_ptr<bst_node> subt){
    if (subt == nullptr) {
        return 0;
    }
    else {
        return 1 + Size(subt->left) + Size(subt->right);
    }
}
```

The above function implements recursion to count the number of nodes in a BST. The time complexity of this operation is $\mathcal{O}(n)$.

The last operation that we will look at is the to vector operation in a BST.

To Vector In BST

To Vector

Below is an example of the implementation of the to vector operation in a BST in the context of C++:

```
/*   ToVector - This function appends values to a vector that are present in a BST
*    Input:
*      subt - A smart pointer of the node that belongs to our BST that we are appending d
*      vec - A vector that integer values are to be appended to from our BST
*    Algorithm:
*      * We first begin by setting the root of our BST to the input parameter "subt"
*      * We then check to see if the root is null, if it is, we return from the function
*      * If the root is not null, then we do the following:
*        * We make a recursive call to the left subtree with "ToVector", to find the left
*        * We then append this data to the input parameter "vec"
*        * We then make a recursive call this time to the right subtree with "ToVector",
*          value of the right subtree
*    Output:
*      This function does not return a value
*/
void BST::ToVector(shared_ptr<bst_node> subt, vector<int>& vec){
    SetRoot(subt);
    if (GetRoot() == nullptr) {
        return;
    }
    else {
        ToVector(subt->left, vec);
        vec.push_back(subt->data);
        ToVector(subt->right, vec);
    }
}
```

The above function appends the data member of the nodes in a BST to a vector with the use of recursion. The time complexity of this operation is $\mathcal{O}(n)$ regardless of the balancing of the tree.

Binary Search Trees (BSTs) are a type of binary tree data structure in which each node has a key/value pair, and the keys in the left subtree are smaller than the key in the node, while the keys in the right subtree are larger. BSTs offer efficient search, insert, and delete operations with an average time complexity of $\mathcal{O}(\log{(n)})$, making them optimal for storing and retrieving data in sorted order. They provide an ordered and hierarchical structure that allows for fast searching and efficient traversal. Additionally, BSTs can be used to implement other data structures like sets, maps, and priority queues.

Linked Lists

Overview

In C++, a linked list is a data structure consisting of a sequence of nodes, where each node contains a value and a pointer/reference to the next node. Unlike arrays and vectors, linked lists do not store elements in contiguous memory locations. Instead, each node in a linked list is dynamically allocated and connected through pointers/references. This allows for efficient insertion and removal of elements at any position in the list, but it requires traversal from the beginning to access specific elements. Linked lists have a flexible size and can expand or shrink as needed by allocating or deallocating nodes, unlike arrays whose size is fixed. However, accessing elements in a linked list requires iterating through the nodes, while arrays and vectors provide direct access using indices.

Common Linked List Operations

Linked lists can be written a number of ways. In the context of this class the linked list was written as a class where the nodes of the linked list was written as a struct. Nodes in linked lists are analgous to elements in an array or vector, the main difference is that nodes contain data (An integer value in the context of this class) and a pointer that points to the next node in the linked list. In doubly linked lists the nodes will contain a pointer that points to the next and the previous nodes in the linked list.

Once a linked list has been created there are a number of operations that can be done on them. Here is a quick summary of some of the linked list operations:

- **Accessing & Modifying:** Accessing or modifying elements in a linked list involves traversing the list by following the node connections until reaching the desired position, and then performing the required operation on the accessed node, such as retrieving or updating its data value.
- **Adding:** Adding an element to a linked list involves creating a new node with the desired data, adjusting the node connections to include the new node in the appropriate position, and updating the necessary pointers to maintain the integrity of the list.
- **Checking For Empty:** Determining if a linked list is empty means checking whether the head node of the list is 'NULL', indicating that there are no elements in the list.
- **Checking Size:** Finding the size of a linked list involves traversing through the list and counting the number of nodes or elements present in the list.
- **Initializing:** Initializing a linked list involves creating the first node, setting its data value, and establishing the necessary connections to indicate the beginning of the list.
- **Removing:** Removing a node from a linked list involves adjusting the node connections to bypass the node to be removed, updating the necessary pointers, and deallocating the memory occupied by the removed node to maintain the integrity of the list.
- **Traversing:** Traversing a linked list means sequentially visiting each node in the list, starting from the head node, and accessing or performing operations on each node until the end of the list is reached.

We will now examine some of these operations in more detail. The first operation that we will look at is accessing and modifying the data in a linked list.

Accessing & Modifying Elements In Linked List

Accessing & Modifying

Below is an example of accessing and modifying elements in a linked list in the context of C++:

```
/*  Contains - This function determines if an element is present inside a linked
*   list
*   Input:
*     data - This is an integer that is being searched for in our linked list
*   Algorithm:
*     First, create a smart pointer called "currPtr" and initialize it to the head
*     of our linked list
*     Initialize the return value "ret" to false, so that if a value is not found
*     we do not need to change it to false
*     Traverse through the linked list with the use of a while loop and check if
*     the current nodes "data" member matches
```

```
*     the input parameter "data"
*       If the two are a match, change the value of "ret" to true, otherwise, leave
*       it as false
*   Output:
*     ret - Boolean value indicating if a value was found in the linked list
*/
bool LinkedList::Contains(int data){
    shared_ptr<node> currPtr = top_ptr_;
    bool ret = false;
    while (currPtr != nullptr) {
        if (currPtr->data == data) {
            ret = true;
        }
        else {}
        currPtr = currPtr->next;
    }
    return ret;
}
```

The above is an example of how we access elements in a linked list. In summary, we start traversing the linked list from the head of the list and check if the current node matches what we are looking for. If it does match, then we modify a boolean value indicating that this element exists in the list. We could simply modify the value of linked list by mutating the data of the node if it passes the if statement. We could return the index of this element with slight modification that keeps track of the position of the current node that we are examining. Conversely, we could find the middle element by counting the total size of the list, and then adding some logic that would allow us to determine what the middle index of the linked list would be.

The next operation that we will examine is adding a node to a linked list.

Adding Elements In A Linked List

Adding Elements

Below is an example of adding a node to the linked list in the context of C++. Particularly, this function adds an element to the end of the linked list (The tail).

```
/*  Append - This function appends a new node to the end of a linked list
*   Input:
*     new_node - A smart pointer to a node object that will be appended to the list
*   Algorithm:
*     We first nitialize the data members of the new node object to appropriate
*     values
*     Then we check if the current linked list is empty:
*       If the linked list is empty, set the top of the linked list to the input
*       parameter "new_node"
*       If the linked list is not empty:
*       We create a new smart pointer called "currPtr" and set it equal to the top
*       of the linked list
*       Then we traverse down the linked list until reaching the last node (When
*       the next node is nullptr):
*       If the next node is not nullptr, move to the next node in the list
*       Once at the end of the linked list, set the next pointer of the current
*       node to "new_node"
*   Output:
*     This function does not return a value
*/
void LinkedList::Append(shared_ptr<node> new_node){
    new_node->data;
    new_node->next = nullptr;
    if (top_ptr_ == nullptr) {
        SetTop(new_node);
    }
    else {
```

```
        shared_ptr<node> currPtr = top_ptr_;
        while (currPtr->next != nullptr) {
            currPtr = currPtr->next;
        }
        currPtr->next = new_node;
    }
}
```

The above function will append a node to the end of the linked list. This function operates on the assumption that the node has already been initialized (Data member has been set). This function can be modified to handle the case where the value that is to be appended is an input parameter of this function. On the contrary, we can add an element at a certain index in the linked list.

Inserting Elements

Below is an example of adding a node to the linked list at a specified index in the context of C++:

```
/*  Insert - This function inserts a new node into a linked list at the specified
*            offset
*   Input:
*     offset - An integer that is the offset at which the new node should be
*              inserted
*     new_node - A smart pointer of object node that is to be inserted into
*                the linked list
*   Algorithm:
*     First we create a new smart pointer called "currPtr" and set it equal to the
*     top of the linked list
*     After that we create a vector called "ptrs" to store the smart pointers to
*     each node in the linked list
*     Then we traverse the linked list and add each node's smart pointer to the
*     "ptrs" vector
*     After that, we check if the offset is zero:
*       If it is zero, set the next pointer of the "new_node" node to the current
*       top_ptr_
*       Set the top_ptr_ to the "new_node" node, making it the new first node in
*       the linked list
*     If the offset is not zero:
*       Reset "currPtr" to the top_ptr_ and iterate through the "ptrs" vector
*       When the current index matches the offset - 1, set the next pointer of
*       "new_node" to currPtr->next
*       Update the next pointer of "currPtr" to "insertPtr", inserting it at the
*       desired offset
*   Output:
*     This function does not return a value
*/
void LinkedList::Insert(int offset, shared_ptr<node> new_node){
    shared_ptr<node> currPtr = top_ptr_;
    vector<shared_ptr<node>> ptrs;
    while (currPtr != nullptr) {
        ptrs.push_back(currPtr);
        currPtr = currPtr->next;
    }
    new_node->data;
    new_node->next = nullptr;
    if (offset == 0) {
        new_node->next = top_ptr_;
        SetTop(new_node);
    }
    else {
    currPtr = top_ptr_;
        for (int i = 0; i < ptrs.size(); i++) {
            if (i == offset - 1) {
            new_node->next = currPtr->next;
```

```
            currPtr->next = new_node;
            }
            else {}
            currPtr = currPtr->next;
        }
    }
}
```

The above function is adding a node to a linked list at a specific index of the linked list. Similar to the previous function that appends a node to the linked list, we can modify this function to append a value at a certain index by having an input parameter that pertains to the desired value to be inserted into the linked list.

The next operation that we will examine is checking if a linked list is empty.

Checking Empty Linked List

Checking Empty

Below is an example of checking if the linked list is empty in the context of C++:

```
bool LinkedList::IsEmpty() {
    return (top_ptr_ == nullptr);
}
```

The above function checks to see if the linked list is empty mainly by seeing if the head of the linked list is empty or not. This check was not necessarily a tenant of the assignment and that is why there are no comments explaining how the function operates.

The next operation that we will examine is checking for the size of a linked list.

Checking Size Of Linked List

Checking Size

Below is an example of checking for the size of a linked list in the context of C++:

```
/*  Size - This function counts how many nodes are present in our linked list
*   Algorithm:
*     We first create a smart pointer called "currPtr" and set it equal to our
*     "top_ptr_" in our linked list
*     We initialize the return value "ret" to zero
*     Then we enter a while loop that increments "ret" by one until it runs into
*     the tail of the list
*     Advance the current pointer object using "currPtr = currPtr->next;"
*   Output:
*     ret - Integer value that represents the number of nodes that are present
*     in our linked list
*/
int LinkedList::Size(){
    shared_ptr<node> currPtr = top_ptr_;
    int ret = 0;
    while (currPtr != nullptr) {
        ret++;
        currPtr = currPtr->next;
    }
    return ret;
}
```

The above function returns the size of a linked list. This function operates by traversing through the linked list and incrementing an integer value that indicates the size of the linked list.

The next operation that we will examine is initializing a node in a linked list.

Initializing A Node In A Linked List

Initializing

Below is an example of initializing a node in a linked list in the context of C++:

```
/*  InitNode - This function takes in an integer data type and assigns that value
*   to a "node" data type's "data" member
*   Input:
*     data - This is an integer data type that is later assigned to a "node" data
*     type's "data" member
*   Algorithm:
*     shared_ptr<node> ret(new node); - This line creates a smart pointer of data
*     type "node" named "ret"
*   Output:
*     ret - After setting "ret"'s data member to the input parameter "data"
*     The "next" data member of the "node" object is initially set to nullptr,
*     indicating the absence of a next node
*/
shared_ptr<node> LinkedList::InitNode(int data){
    shared_ptr<node> ret(new node);
    ret->data = data;
    return ret;
}
```

The above function initializes a node in a linked list to a specific value. This function does not necessarily update the pointer to the next node in the linked list but that is done in the other functions that have been previously defined.

The next operation that we will examine is removing a node from a linked list.

Removing A Node From A Linked List

Removing

The below function is an example of removing a node from a linked list in the context of C++:

```
/*  Remove - This function removes a node from a linked list at the specified
*            offset
*   Input:
*     offset - An integer that is the offset of the node to be removed from our
*              linked list
*   Algorithm:
*     First we create a new smart pointer called "currPtr" and set it equal to the
*     top of the linked list
*     Then we create a vector called "ptrs" to store the smart pointers to each
*     node in the linked list
*     After that we iterate through the linked list and add each node's smart
*     pointer to the "ptrs" vector
*     Then we heck if the offset is zero.
*       If it is zero, set the next pointer of the top node to the node after it
*     Otherwise, reset "currPtr" to the top node and iterate through the "ptrs"
*     vector
*       When the current index matches the offset - 1, set the next pointer of
*       "currPtr" to the node after it
*       Break out of the for loop to avoid segmentation faults
*     Output:
*       This function does not return a value.
*/
void LinkedList::Remove(int offset){
    shared_ptr<node> currPtr = top_ptr_;
    vector<shared_ptr<node>> ptrs;
    while (currPtr != nullptr) {
        ptrs.push_back(currPtr);
        currPtr = currPtr->next;
```

```
    }
    if (offset == 0) {
        SetTop(top_ptr_->next);
    }
    else {
        currPtr = top_ptr_;
        for (int i = 0; i < ptrs.size(); i++) {
            if (i == offset - 1) {
                currPtr->next = currPtr->next->next;
                break;
            }
            else {}
                currPtr = currPtr->next;
        }
    }
}
```

The above function removes a node from a linked list at a specified offset of the linked list. We can remove a specific node from a linked list by searching for if the node exists in the linked list and then deleting the node from the linked list. This implementation requires a separate function that will do the removing of the node from the linked list.

The final common operation of the linked list is traversing the linked list. This operation is required for a lot of functions that were previously defined in the linked list. This is done by using a while loop that runs until the current pointer in the linked list is null or some other condition that should be met. We now will look at some extra operations that can be utilized in a linked list.

Extra Linked List Operations

There are some extra operations that can be done on a linked list. These involve operations from reversing a linked list to splitting a linked list. Here is a comprehensive list of extra operations in linked lists:

- **Concatenating:** Concatenating a linked list involves combining two or more linked lists into a single linked list, where the last node of the first list is connected to the first node of the second list, and so on, creating a continuous sequence of nodes.
- **Reversing:** Reversing a linked list involves changing the direction of the next pointers in each node, effectively flipping the order of the elements.
- **Sorting:** Sorting a linked list involves rearranging the nodes in a specific order, such as ascending or descending, based on the values contained in each node.
- **Splitting:** Splitting a linked list involves dividing a single linked list into two separate linked lists, typically based on a given condition or position.

We now can examine these operations in more detail. The first extra operation that we are looking at is the concatenating of a linked list.

Concatenating A Linked List

Concatenating

Below is an example of concatenating a linked list in the context of C++:

```
std::shared_ptr<node> concatenateLists(std::shared_ptr<node> head1,
                                       std::shared_ptr<node> head2) {
    if (head1 == nullptr)
        return head2;
    if (head2 == nullptr)
        return head1;
    std::shared_ptr<node> current = head1;
    while (current->next != nullptr) {
        current = current->next;
    }
    current->next = head2;
```

```
    return head1;
}
```

The above function takes in two heads of linked lists. It then traverses the linked list looking for the tail of the first list. It then updates the tail of the first linked list and has it point to the head of the second linked list.

The next extra operation that we are going to look at is reversing a linked list.

Reversing A Linked List

Reversing

Below is an example of reversing a linked list in the context of C++:

```
std::shared_ptr<node> reverseList(std::shared_ptr<node> head) {
    std::shared_ptr<node> current = head;
    std::shared_ptr<node> previous = nullptr;
    std::shared_ptr<node> next = nullptr;
    while (current != nullptr) {
        next = current->next;
        current->next = previous;
        previous = current;
        current = next;
    }
    return previous;
}
```

The above function reverses the nodes in a linked list. This is done by keeping track of the current, next, and previous nodes in the linked list. This entire operation could be done a lot easier if the linked list in question was a doubly linked list. But this function works on a singly linked list.

The next extra operation that we are going to look at is sorting a linked list. There are a multitude of ways that this can be accomplished, by choosing different sorting algorithms. The easiest way this is done is with the use of a bubble sort.

Sorting A Linked List

Sorting

Below is an example of sorting a linked list with the use of a bubble sort in the context of C++:

```
void bubbleSort(std::shared_ptr<node> head) {
    if (head == nullptr || head->next == nullptr) {
        return;
    }
    bool swapped;
    std::shared_ptr<node> current;
    std::shared_ptr<node> last = nullptr;
    do {
        swapped = false;
        current = head;
        while (current->next != last) {
            if (current->data > current->next->data) {
                // Swap the nodes
                std::swap(current->data, current->next->data);
                swapped = true;
            }
            current = current->next;
        }
        last = current;
    } while (swapped);
}
```

The above function sorts the data in a linked list with the use of a bubble sort algorithm. This algorithm does not require one to update the pointers of the nodes as it is simply mutating the data members of the nodes. One can sort the elements of the linked list in descending order by just changing the if statement that is present in the while loop.

The last extra operation that we are going to look at is splitting a linked list at a specific position.

Splitting A Linked List

Splitting

Below is an example of splitting a linked list in the context of C++:

```
std::pair<std::shared_ptr<node>, std::shared_ptr<node>>
        splitList(std::shared_ptr<node> head, int splitPosition) {
    std::shared_ptr<node> splitPtr = nullptr;
    std::shared_ptr<node> secondHead = nullptr;
    std::shared_ptr<node> current = head;
    int position = 0;
    // Traverse until the split position
    while (current != nullptr && position < splitPosition - 1) {
        current = current->next;
        position++;
    }
    if (current != nullptr) {
        splitPtr = current->next;
        current->next = nullptr;  // Terminate the first list
        secondHead = splitPtr;
    }
    return {head, secondHead};
}
```

The above function splits a linked list at a specific index of the original linked list. This is done by returning a pair of head nodes after the function is done executing. In the function we iterate until we reach the specified index or if the index happens to be out of bounds. We then assign the head of the second linked list by grabbing the next node after the index position.

Other Implementations Of Linked Lists

There are many other implentations for linked lists. Some implementations of linked lists are the following:

- **Circular Linked List:** - Implementing a linked list as a circular linked list involves connecting the last node of the list to the first node, creating a circular structure.
- **Doubly Linked List:** - Implementing a linked list as a doubly linked list involves adding an additional pointer in each node that points to the previous node.
- **Stack Or Queue:** - Implementing a linked list as a stack or queue involves utilizing the respective operations of push/pop for a stack or enqueue/dequeue for a queue to insert or remove elements at one end of the linked list.

We now will look at these implementations one by one. The first implementation that we will look at is circular linked lists.

Circular Linked List

Circular Linked Lists

Below is an example of how to implement a circular linked list in the context of C++:

```
struct Node {
    int data;
    std::shared_ptr<Node> next;
    Node(int val) : data(val), next(nullptr) {}
};

void traverseCircularLinkedList(const std::shared_ptr<Node>& head) {
    if (head == nullptr) {
        return;
```

```
    }

    std::shared_ptr<Node> current = head;

    do {
        std::cout << current->data << " ";
        current = current->next;
    } while (current != head);

    std::cout << std::endl;
}

int main() {
    // Create a circular linked list with three nodes: 1 -> 2 -> 3 -> (back to 1)
    std::shared_ptr<Node> head = std::make_shared<Node>(1);
    std::shared_ptr<Node> second = std::make_shared<Node>(2);
    std::shared_ptr<Node> third = std::make_shared<Node>(3);

    head->next = second;
    second->next = third;
    third->next = head;  // Make it circular

    // Traverse the circular linked list
    traverseCircularLinkedList(head);  // Output: 1 2 3

    return 0;
}
```

The above is an example of how to make a linked list circular. The crux of making a linked list circular is to have the last node of the list point to the head of the list. This is done inside the main function above.

Next, we look at how to implement a doubly linked list.

Doubly Linked List

Doubly Linked Lists

Below is an example of doubly linked lists in the context of C++:

```
#include <iostream>
#include <memory>

struct Node {
    int data;
    std::shared_ptr<Node> prev;
    std::shared_ptr<Node> next;
    Node(int val) : data(val), prev(nullptr), next(nullptr) {}
};

void traverseDoublyLinkedListForward(const std::shared_ptr<Node>& head) {
    std::shared_ptr<Node> current = head;

    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }

    std::cout << std::endl;
}

void traverseDoublyLinkedListBackward(const std::shared_ptr<Node>& tail) {
    std::shared_ptr<Node> current = tail;
```

```
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->prev;
    }

    std::cout << std::endl;
}

int main() {
    // Create a doubly linked list with three nodes: 1 <-> 2 <-> 3
    std::shared_ptr<Node> head = std::make_shared<Node>(1);
    std::shared_ptr<Node> second = std::make_shared<Node>(2);
    std::shared_ptr<Node> third = std::make_shared<Node>(3);

    head->next = second;
    second->prev = head;
    second->next = third;
    third->prev = second;

    // Traverse the doubly linked list forward and backward
    traverseDoublyLinkedListForward(head);  // Output: 1 2 3
    traverseDoublyLinkedListBackward(third);  // Output: 3 2 1

    return 0;
}
```

The above is an example of how to create a doubly linked list. In this example, the main difference between our linked list class that we created previously is that the node structure has a previous pointer as well as a next pointer. This in essence is how a doubly linked list is created.

Lastly, we look at how to implement a stack and queue with the use of a linked list.

Stacks & Queues With Linked Lists

Stacks & Queues

Below is an example of how to implement a linked list with the stack and queue data structure. This is first done by looking at the structure of the nodes in the linked list in the context of C++:

```
struct Node {
    int data;
    std::shared_ptr<Node> next;
    Node(int val) : data(val), next(nullptr) {}
};
```

We then move on to looking how we implement the stack data structure with this node structure:

```
class Stack {
    private:
        std::shared_ptr<Node> top;

    public:
        Stack() : top(nullptr) {}

        void push(int val) {
            std::shared_ptr<Node> newNode = std::make_shared<Node>(val);
            newNode->next = top;
            top = newNode;
        }

        void pop() {
            if (top) {
                std::shared_ptr<Node> temp = top;
```

```
                top = top->next;
                temp.reset();
            }
        }

        int peek() {
            if (top) {
                return top->data;
            }
            throw std::runtime_error("Stack is empty.");
        }

        bool isEmpty() {
            return top == nullptr;
        }
};
```

Next, we look at the queue data structure to see how it would be implemented:

```
class Queue {
    private:
        std::shared_ptr<Node> front;
        std::shared_ptr<Node> rear;

    public:
        Queue() : front(nullptr), rear(nullptr) {}

        void enqueue(int val) {
            std::shared_ptr<Node> newNode = std::make_shared<Node>(val);
            if (!front) {
                front = newNode;
                rear = newNode;
            } else {
                rear->next = newNode;
                rear = newNode;
            }
        }

        void dequeue() {
            if (front) {
                std::shared_ptr<Node> temp = front;
                front = front->next;
                temp.reset();
            }
        }

        int peek() {
            if (front) {
                return front->data;
            }
            throw std::runtime_error("Queue is empty.");
        }

        bool isEmpty() {
            return front == nullptr;
        }
};
```

And lastly, we look at how these would be implemented:

```
int main() {
    Stack stack;
```

```
    stack.push(10);
    stack.push(20);
    stack.push(30);

    while (!stack.isEmpty()) {
        std::cout << stack.peek() << " ";
        stack.pop();
    }
    // Output: 30 20 10

    Queue queue;
    queue.enqueue(5);
    queue.enqueue(15);
    queue.enqueue(25);

    while (!queue.isEmpty()) {
        std::cout << queue.peek() << " ";
        queue.dequeue();
    }
    // Output: 5 15 25

    return 0;
}
```

The above is only one way this can be achieved. There are numerous different ways this outcome can be produced.

Overall, linked lists are a very powerful data structure in object oriented programming. They allow for easy traversal, insertion, and deletion of elements. The biggest downside to linked lists is that someone cannot easily access elements inside the linked list without traversing the list itself. Doubly linked lists allow for easy traversal in each direction but require additional logic to update parameters that are found in the structure of the linked list.

Vectors

C++ Vector Overview

In C++, a vector is a dynamic array-like container provided by the Standard Template Library (STL). It is a sequence container that can dynamically resize itself to accommodate elements. Unlike an array, which has a fixed size determined at compile time, a vector can grow or shrink in size as elements are added or removed. Vectors offer various advantages over arrays, such as automatic memory management, the ability to easily change their size, and built-in functions for common operations like inserting, erasing, and accessing elements. Vectors provide similar functionality to arrays but with added flexibility and convenience.

How Do You Declare And Initialize A Vector With Values?

Initializing A Vector

The way we initialize a vector in C++ with values is like the following:

```
#include <vector>

int main() {
    // Declare and initialize a vector of integers
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Declare and initialize a vector of strings
    std::vector<std::string> names = {"Alice", "Bob", "Charlie"};

    // Declare an empty vector
    std::vector<double> values;

    return 0;
}
```

Vector Operations In C++

Here are some common vector operations in C++. First, we look at how we access elements in a vector.

Accessing Vector Elements

Accessing Elements

Here is how someone accesses elements in a vector:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Accessing elements using the indexing operator []
int firstElement = numbers[0];  // Access the first element (1)
int thirdElement = numbers[2];  // Access the third element (3)

// Accessing elements using the at() member function
int secondElement = numbers.at(1);  // Access the second element (2)
int fourthElement = numbers.at(3);  // Access the fourth element (4)
```

Here is how someone adds elements to a vectors.

Adding Vector Elements

Adding To End

Adding an element at the end of a vector with **push_back()**:

```
    std::vector<int> numbers = {1, 2, 3, 4, 5};
```

```
    // Add the number 6 to the end of the vector
    numbers.push_back(6);

    // Print the modified vector
    for (int i = 0; i < numbers.size(); i++) {
        std::cout << numbers[i] << std::endl;
    }
```

Here is how vectors can be cleared.

Clearing Vectors

Clearing All Elements

Clearing all the elements from a vector with **clear()**:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Remove all elements from the vector
numbers.clear();

// Print the modified vector
for (int i = 0; i < numbers.size(); i++) {
    std::cout << numbers[i] << std::endl;
}
```

If you want to find the capacity of a vector, this is how its done.

Finding Capacity Of Vector

Finding Capacity

Finding the capacity of the vector with **capacity()**:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Find the capacity of the vector
int capacity = numbers.capacity();

// Print the capacity of the vector
std::cout << "The capacity of the vector is: " << capacity << std::endl;
```

To find the size of vector, this is how its done.

Finding Size Of Vector

Finding Size

Finding the size of the vector with **size()**:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Find the size of the vector
int size = numbers.size();

// Print the size of the vector
std::cout << "The size of the vector is: " << size << std::endl;
```

Next, we look at how we modify elements in a vector.

Modifying Vector Elements

Modifying Elements

Here is how someone modifies elements in a vector:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Modifying elements using the indexing operator []
numbers[0] = 10;  // Modify the value of the first element to 10
numbers[2] = 30;  // Modify the value of the third element to 30

// Modifying elements using the at() member function
numbers.at(1) = 20;  // Modify the value of the second element to 20
numbers.at(3) = 40;  // Modify the value of the fourth element to 40
```

There are multiple ways to remove an element at an index of a vector, here is one.

Removing Element Of Vector

Removing Element At Index

Removing an element at an index of a vector with **erase()**:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Remove the element at index 2 from the vector
int element_to_remove = numbers[2];
numbers.erase(numbers.begin() + 2);

// Print the modified vector
for (int i = 0; i < numbers.size(); i++) {
    std::cout << numbers[i] << std::endl;
}
```

Another way to remove an element is to remove the last element of a vector.

Removing Last Element Of Vector

Removing Last Element

Removing the last element from a vector with **pop_back()**:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Remove the last element from the vector
int last_element = numbers.back();
numbers.pop_back();

// Print the modified vector
for (int i = 0; i < numbers.size(); i++) {
    std::cout << numbers[i] << std::endl;
}
```

Vectors can be sorted as well, here is how this is done.

Sorting Vectors

Sorting

Vectors can be sorted with sorting algorithms or the built in STL function **sort()**:

```
std::vector<int> numbers = {5, 2, 8, 1, 9};
```

```cpp
// Sort the vector in ascending order
std::sort(numbers.begin(), numbers.end());

// Print the sorted vector
for (const auto& num : numbers) {
    std::cout << num << " ";
}
std::cout << std::endl;
```

Here is a summary of some common vector operations.

- Accessing Elements:
  - Using the indexing operator [] to access individual elements by their index.
  - Using the **at()** member function to access elements with bounds checking.
  - Using iterators to traverse the vector and access elements.
- Iterating & Traversing:
  - Using range-based **for** loops or iterators to iterate over the elements of the vector.
  - Using algorithms like **for_each()**, **transform()**, or **accumulate()** from the **algorithm** library to perform operations on each element.
- Modifying Elements:
  - Using the assignment operator = to modify individual elements directly.
  - Using the **push_back()** member function to add elements at the end of the vector.
  - Using the **pop_back()** member function to remove the last element from the vector.
  - Using the **insert()** member function to insert elements at a specified position.
  - Using the **erase()** member function to remove elements at a specified position or range.
- Size & Capacity:
  - Using the **size()** member function to get the current number of elements in the vector.
  - Using the **empty()** member function to check if the vector is empty.
  - Using the **resize()** member function to change the size of the vector.
  - Using the **reserve()** member function to allocate memory for a specified number of elements.
- Sorting & Manipulating Order:
  - Using the **sort()** function from the **algorithm** library to sort the elements in the vector.
  - Using the **reverse()** function from the **algorithm** library to reverse the order of elements.
  - Using algorithms like **find()**, **count()**, or **binary_search()** from the **algorithm** library to search for elements.

Benefits Of Vectors Over Arrays

There are many benefits to using a vector over an array. Some examples of this are dynamic size, automatic memory management, convenient functions, range checking, and copy and assignment. These are not all of the advantages, rather just some commond examples. Diving into these examples a little bit more:

- **Automatic Memory Management:** Vectors handle memory allocation and deallocation automatically. When elements are added or removed, the vector manages the underlying memory for you. Arrays, on the other hand, require manual memory management.
- **Convenient Functions:** Vectors provide a range of built-in functions that make working with elements more convenient. For example, vectors have functions like **push_back()** to add elements at the end, **pop_back()** to remove elements from the end, and **insert()** to insert elements at specific positions. Arrays lack such built-in functions, requiring manual manipulation of elements.
- **Copy and Assignment:** Vectors can be easily copied or assigned to another vector using the assignment operator (=) or the copy constructor. This simplifies the process of creating copies or working with multiple vectors. Arrays do not have built-in copy or assignment mechanisms, requiring manual copying of each element.

- **Dynamic Size:** Vectors have a flexible and resizable size, allowing elements to be added or removed easily. In contrast, arrays have a fixed size determined at compile-time, making it difficult to change their size dynamically.
- **Range Checking:** Vectors perform bounds checking to ensure that accessing elements stays within the valid range. This helps prevent accessing elements outside the vector's size, which can lead to runtime errors. Arrays do not perform such range checking, allowing access to elements beyond the declared size, which can lead to undefined behavior.

Overall, vectors provide more flexibility, convenience, and safety compared to arrays in terms of size management, memory handling, and operations on elements.

C++ Vector Functions Recap

Here is a quick summary of vector functions in C++:

- **back():** The 'back()' method in C++ returns a reference to the last element in a vector.
- **capacity():** The 'capacity()' method in C++ returns the current storage capacity of a vector, which represents the maximum number of elements that the vector can hold without requiring reallocation of memory.
- **clear():** The 'clear()' method in C++ removes all elements from a vector, effectively emptying it.
- **erase():** The 'erase()' method in C++ removes one or a range of elements from a vector, based on the specified position or iterator.
- **find():** The 'find()' function in C++ searches for a specified element in a vector and returns an iterator pointing to the first occurrence of the element.
- **front():** The 'front()' method in C++ returns a reference to the first element in a vector.
- **pop_back():** The 'pop_back()' method in C++ removes the last element from a vector.
- **push_back():** The 'push_back()' method in C++ adds an element to the end of a vector.
- **resize():** The 'resize()' method in C++ changes the size of the vector, either increasing or decreasing it.
- **size():** The 'size()' method in C++ returns the number of elements in a vector, representing the current size of the vector.