Data Structure and Algorithms Notes

Contents

[Linked Lists: 2](#_Toc184251936)

[Constructor: 3](#_Toc184251937)

[Destructor: 3](#_Toc184251938)

[Traversal: 4](#_Toc184251939)

[Searching: 4](#_Toc184251940)

[Finding Length: 5](#_Toc184251941)

[Insertion: 6](#_Toc184251942)

[Insertion at Beginning: 6](#_Toc184251943)

[Insertion at End: 7](#_Toc184251944)

[Insertion at Specific Position: 8](#_Toc184251945)

[Deletion: 9](#_Toc184251946)

[Deletion at Beginning: 9](#_Toc184251947)

[Deletion at the End 10](#_Toc184251948)

[Deletion of Specific Position: 12](#_Toc184251949)

[Stacks: 15](#_Toc184251950)

[Constructor: 17](#_Toc184251951)

[Destructor: 17](#_Toc184251952)

[Push() – Insert element into stack: 18](#_Toc184251953)

[Pop() – Remove element from top of stack: 19](#_Toc184251954)

[getTop(ItemType &item) – Returns top element from stack: 21](#_Toc184251955)

[isEmpty() – Returns true if empty, else false 22](#_Toc184251956)

[append() – Adds to the end of the stack 22](#_Toc184251957)

[Queues: 24](#_Toc184251958)

[Constructor: 25](#_Toc184251959)

[Enqueue(ItemType &item): 25](#_Toc184251960)

[Dequeue(): 26](#_Toc184251961)

[Dequeue(ItemType& Item): 27](#_Toc184251962)

[getFront(ItemType& Item): 27](#_Toc184251963)

[isEmpty(): 28](#_Toc184251964)

[Destructor: 28](#_Toc184251965)

[Hash Tables: 29](#_Toc184251966)

[Constructor: 30](#_Toc184251967)

[Add(KeyType, ItemType): 30](#_Toc184251968)

[Remove(KeyType): 31](#_Toc184251969)

[get(KeyType): 32](#_Toc184251970)

[isEmpty(): 33](#_Toc184251971)

[getLength() 33](#_Toc184251972)

[Destructor() 33](#_Toc184251973)

# Linked Lists:

A diagram of a number

Description automatically generated

Characteristics:

* Each node has a single value and reference to the next node in the list
* List has a head, which is a reference to the first node in the list. We can access all items of a list using head node, sometimes a tail (reference to the last node) is also used.
* Nodes are not stored in contiguous block of memory, but each node holds address of the next node in the list. Accessing elements in a singly linked list requires traversing from the head to desired node, as there is NO direct access to a specific node in memory

Advantages:

* Insertion and Deletion take O(1) Time, in an array it is O(n)
* Linked list is more space efficient, does not waste storage due to dynamic memory allocation
* Size of the list is not fixed, able to grow as large as possible

Disadvantages:

* Slow access time: Traverse the linked list to find element which is O(n) operation
* Pointers & References: Complex to understand
* Higher overhead: Each node in a link list requires more memory to store reference to next node
* Cache Inefficiency: Due to memory not being contiguous

**Basic Operations:**

1. Traversal
2. Searching
3. Length
4. Insertion:
   1. Insert at the beginning
   2. Insert at the end
   3. Insert at a specific position
5. Deletion:
   1. Delete from the beginning
   2. Delete from the end
   3. Delete a specific node

## Constructor:

Just set head to nullptr;

 // Constructor to initialize an empty linked list

    LinkedList() : head(nullptr) {}

## Destructor:

School method insist on using while loop (when head != nullptr) and call the function removeLastNode. However, this is inefficient because it traverses the entire list every loop.

The most efficient way is to code a new removeAllNodes function:

// LinkedList destructor

LinkedList::~LinkedList() {

    deleteAllNodes(); // Call the helper function to delete all nodes

}

void LinkedList::deleteAllNodes() {

    Node\* current = head;

    while (current != nullptr) {

        Node\* next = current->next;

        delete current;

        current = next;

    }

    head = nullptr;

}

## Traversal:

   Time Complexity:

        O(n) - Where n is the number of nodes in the list, as it visits each node once.

    Space Complexity:

        O(1) - Uses a constant amount of extra space.

* Initialize a pointer current to the head of the list.
* Use a while loop to iterate through the list until the current pointer reaches NULL.
* Inside the loop, print the data of the current node and move the current pointer to the next node.
* void traverseLinkedList(Node\* head) {
* Node\* current = head; // Initialize 'current' to start at the head of the list
* // Iterate through the list until the end is reached
* while(current != nullptr) {
* cout << current->data << " "; // Output the data of the current node
* current = current->next;       // Move to the next node in the list
* }
* cout << endl; // Print a newline character after traversal for better output formatting
* }

## Searching:

    Time Complexity:

        O(n) - In the worst case, where n is the number of nodes, the function may need to traverse the entire list.

    Space Complexity:

        O(1) - Uses a constant amount of extra space.

* Traverse the Linked List starting from the head.
* Check if the current node's data matches the target value.
  + If a match is found, return true.
* Otherwise, Move to the next node and repeat steps 2.
* If the end of the list is reached without finding a match, return false.

bool searchLinkedList(Node\* head, int value) {

    Node\* current = head; // Initialize 'current' to start at the head of the list

    // Traverse the list to search for the value

    while (current != nullptr) {

        if (current->data == value) { // Check if current node contains the target value

            return true;              // Value found; return true

        }

        current = current->next;      // Move to the next node in the list

    }

    return false; // Value not found after complete traversal; return false

}

## Finding Length:

  Time Complexity:

        O(n) - Where n is the number of nodes, as it traverses each node once.

    Space Complexity:

        O(1) - Uses a constant amount of extra space.

* Initialize a counter **length**to 0.
* Start from the head of the list, assign it to current.
* Traverse the list:
  + Increment **length**for each node.
  + Move to the next node (**current = current->next**).
* Return the final value of **length**.

int findLength(Node\* head) {

    int length = 0;        // Initialize counter to track the number of nodes

    Node\* current = head;  // Initialize 'current' to start at the head of the list

    // Traverse the list to count the nodes

    while (current != nullptr) {

        length++;                    // Increment the counter for each node

        current = current->next;     // Move to the next node in the list

    }

    return length; // Return the total number of nodes in the list

}

## Insertion:

### Insertion at Beginning:

    Time Complexity:

        O(1) - Constant time insertion at the beginning.

    Space Complexity:

        O(1) - Only a new node is created regardless of list size.

A diagram of a line of pills

Description automatically generated

* Create a new node with the given value.
* Set the **next**pointer of the new node to the current head.
* Move the head to point to the new node.
* Return the new head of the linked list.

Node\* insertAtBeginning(Node\* head, int value) {

    Node\* newNode = new Node(value); // Create a new node with the specified value

    newNode->next = head;            // Link the new node to the current head of the list

    head = newNode;                  // Update 'head' to point to the new node, making it the new head

    return head;                     // Return the updated head of the list

}

### Insertion at End:

    Time Complexity:

        O(n) - In the worst case, where n is the number of nodes, the function traverses the entire list to find the last node.

    Space Complexity:

        O(1) - Only a new node is created regardless of list size.

A diagram of a line of pills

Description automatically generated

* Create a new node with the given value.
* Check if the list is empty:
  + If it is, make the new node the head and return.
* Traverse the list until the last node is reached.
* Link the new node to the current last node by setting the last node's next pointer to the new node.

Node\* insertAtEnd(Node\* head, int value) {

    Node\* newNode = new Node(value); // Create a new node with the specified value

    if (head == nullptr) {           // Check if the list is empty

        return newNode;              // If empty, the new node becomes the head of the list

    }

    Node\* current = head;            // Initialize 'current' to start at the head of the list

    // Traverse the list to find the last node

    while (current->next != nullptr) {

        current = current->next;     // Move to the next node in the list

    }

    current->next = newNode;         // Link the last node to the new node, effectively adding it to the end

    return head;                     // Return the head of the list (unchanged)

}

### Insertion at Specific Position:

    Time Complexity:

        O(n) - In the worst case, where n is the number of nodes, the function may need to traverse up to position-1 nodes.

    Space Complexity:

        O(1) - Only a new node is created regardless of list size.

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

We mainly find the node after which we need to insert the new node. If we encounter a NULL before reaching that node, it means that the given position is invalid.

Node\* insertAtPosition(Node\* head, int position, int data) {

    // Validate that the position is a positive integer

    if (position < 1) {

        std::cerr << "Error: Invalid position " << position << ". Position must be >= 1." << std::endl;

        return head; // Return the original head without making changes

    }

    Node\* newNode = new Node(data); // Create a new node with the specified data

    // If the position is 1, insert the new node at the beginning

    if (position == 1) {

        newNode->next = head; // Link the new node to the current head

        return newNode;       // The new node becomes the new head of the list

    }

    Node\* current = head;      // Initialize 'current' to start at the head of the list

    int currentPosition = 1;   // Initialize a counter to track the current position

    // Traverse the list to find the node just before the desired insertion position

    while (currentPosition < position - 1 && current != nullptr) {

        current = current->next; // Move to the next node in the list

        currentPosition++;       // Increment the position counter

    }

    // After traversal, check if 'current' is nullptr, indicating an out-of-bounds position

    if (current == nullptr) {

        std::cerr << "Error: Position " << position << " is out of bounds." << std::endl;

        delete newNode; // Delete the new node to prevent a memory leak

        return head;    // Return the original head without making changes

    }

    // Insert the new node at the desired position

    newNode->next = current->next; // Link the new node to the next node in the list

    current->next = newNode;       // Link the previous node to the new node

    return head; // Return the head of the list (unchanged)

}

## Deletion:

### Deletion at Beginning:

    Time Complexity:

        O(1) - Constant time deletion from the beginning.

    Space Complexity:

        O(1) - Uses a constant amount of extra space.

A diagram of a flowchart

Description automatically generated

* Check if the head is **NULL**.
  + If it is, return **NULL**(the list is empty).
* Store the current head node in a temporary variable **temp**.
* Move the head pointer to the next node.
* Delete the temporary node.
* Return the new head of the linked list.

Node\* deleteFromBeginning(Node\* head) {

    if (head == nullptr) { // Check if the list is empty

        std::cerr << "Error: Cannot delete from an empty list." << std::endl;

        return head; // Return nullptr as the list is already empty

    }

    Node\* temp = head;    // Temporarily store the current head node

    head = head->next;    // Update 'head' to point to the next node in the list

    delete temp;          // Delete the old head node to free memory

    return head;          // Return the new head of the list

}

### Deletion at the End

    Time Complexity:

        O(n) - In the worst case, where n is the number of nodes, the function traverses the entire list to find the second last node.

    Space Complexity:

        O(1) - Uses a constant amount of extra space.

A diagram of a deletion

Description automatically generated

* Check if the head is **NULL**.
  + If it is, return NULL (the list is empty).
* Check if the head's **next**is **NULL**(only one node in the list).
  + If true, delete the head and return **NULL**.
* Traverse the list to find the second last node (**second\_last**).
* Delete the last node (the node after **second\_last**).
* Set the **next**pointer of the second last node to **NULL**.
* Return the head of the linked list.

Node\* removeLastNode(Node\* head) {

    if (head == nullptr) { // Check if the list is empty

        std::cerr << "Error: Cannot delete from an empty list." << std::endl;

        return head; // Return nullptr as the list is already empty

    }

    if (head->next == nullptr) { // Check if there is only one node in the list

        delete head;              // Delete the single node

        return nullptr;           // Return nullptr as the list is now empty

    }

    Node\* second\_last = head; // Initialize 'second\_last' to start at the head of the list

    // Traverse the list to find the second last node

    while (second\_last->next->next != nullptr) {

        second\_last = second\_last->next; // Move 'second\_last' to the next node

    }

    // After traversal, 'second\_last->next' is the last node

    delete second\_last->next; // Delete the last node to free memory

    second\_last->next = nullptr; // Set 'second\_last->next' to nullptr to indicate the new end of the list

    return head; // Return the head of the list (unchanged)

}

### Deletion of Specific Position:

   Time Complexity:

        O(n) - In the worst case, where n is the number of nodes, the function may need to traverse up to position-1 nodes.

    Space Complexity:

        O(1) - Uses a constant amount of extra space.

A diagram of a diagram

Description automatically generated

* Check if the list is empty or the position is invalid, return if so.
* If the head needs to be deleted, update the head and delete the node.
* Traverse to the node before the position to be deleted.
* If the position is out of range, return.
* Store the node to be deleted.
* Update the links to bypass the node.
* Delete the stored node.

Node\* deleteAtPosition(Node\* head, int position)

{

    // Step 1: Check if the list is empty or the position is invalid

    if (head == nullptr) {

        cerr << "Error: Cannot delete from an empty list." << endl;

        return head; // Return the original head as the list is empty

    }

    if (position < 1) {

        cerr << "Error: Invalid position " << position << ". Position must be >= 1." << endl;

        return head; // Return the original head as the position is invalid

    }

    // Step 2: If the head needs to be deleted

    if (position == 1) {

        Node\* temp = head;    // Store the current head in a temporary pointer

        head = head->next;    // Update 'head' to point to the next node in the list

        delete temp;          // Delete the old head node to free memory

        return head;          // Return the new head of the list

    }

    // Step 3: Traverse to the node before the position to be deleted

    Node\* current = head; // Initialize 'current' to start at the head of the list

    // Loop to reach the (position - 1)th node

    for (int i = 1; i < position - 1 && current != nullptr; i++) {

        current = current->next; // Move 'current' to the next node

    }

    // Step 4: Check if the position is out of range

    if (current == nullptr || current->next == nullptr) {

        cerr << "Error: Position " << position << " is out of bounds." << endl;

        return head; // Return the original head as the position is invalid

    }

    // Step 5: Delete the node at the specified position

    Node\* temp = current->next;          // Store the node to be deleted

    current->next = current->next->next; // Bypass the node to be deleted

    delete temp;                         // Delete the target node to free memory

    // Step 6: Return the head of the linked list

    return head;

}

# Stacks:

A diagram of a structure

Description automatically generated

Stack is a linear data structure based on LIFO(Last In First Out) principle in which the insertion of a new element and removal of an existing element takes place at the same end represented as the top of the stack.

To implement the stack, it is required to maintain the pointer to the top of the stack , which is the last element to be inserted because we can access the elements only on the top of the stack.

**Advantages of Array Implementation:**

* Easy to implement.
* Memory is saved as pointers are not involved.

**Disadvantages of Array Implementation:**

* It is not dynamic i.e., it doesn’t grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
* The total size of the stack must be defined beforehand.

**Advantages of Linked List implementation:**

* The linked list implementation of a stack can grow and shrink according to the needs at runtime.
* It is used in many virtual machines like JVM.

**Disadvantages of Linked List implementation:**

* Requires extra memory due to the involvement of pointers.
* Random accessing is not possible in stack.

**Advantages of Stack Data Structure:**

1. **Simplicity:**Stacks are a simple and easy-to-understand data structure, making them suitable for a wide range of applications.
2. **Efficiency:**Push and pop operations on a stack can be performed in constant time **(O(1))**, providing efficient access to data.
3. **Last-in, First-out (LIFO):**Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed. This behaviour is useful in many scenarios, such as function calls and expression evaluation.
4. **Limited memory usage:**Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

**Disadvantages of Stack Data Structure:**

* **Limited access:**Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
* **Potential for overflow:**If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
* **Not suitable for random access:**Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
* **Limited capacity:**Stacks have a fixed capacity, which can be a limitation if the number of elements that need to be stored is unknown or highly variable.

**Basic Operations on Stack Data Structure (Pointers only):**

To make manipulations in a stack, there are certain operations provided to us.

1. **Constructor** to create an empty stack
2. **push(ItemType &item):bool**to insert an element into the stack
3. **pop():bool**to remove an element from the stack
4. **getTop(ItemType &item)**Returns the top element of the stack.
5. **isEmpty()**returns true if stack is empty else false.
6. **append()** insert an element into the bottom of the stack
7. **Destructor** to destroy a stack

## Constructor:

Set head/root to nullptr;

 // Constructor to initialize an empty stack

    Stack() : root(nullptr) {}

## Destructor:

Delete each node in the stack

    // Destructor to free all nodes in the stack

    ~Stack() {

        while (root) {

            StackNode\* temp = root;

            root = root->next;

            delete temp;  // Delete each node in the stack

        }

    }

## Push() – Insert element into stack:

Pushes an element into the top of the stack

No need to check if full because pointers-based, only arrays have max size

Time Complexity: O(1) - Constant time as it only adds a single node at the top

Space Complexity: O(1) - Only a single pointer is allocated for the new node

A diagram of a stack

Description automatically generated

bool Stack::push(ItemType item)

{

    Node\* newNode = new Node;

    if (newNode == nullptr) // Check if memory allocation failed

    {

        return false;

    }

    newNode->item = item;

    newNode->next = topNode;

    topNode = newNode;

    return true;

}

## Pop() – Remove element from top of stack:

Removes the top most element from the stack

Time Complexity: O(1) - Constant time as it only removes a single node from the top

 Space Complexity: O(1) - Only temporary storage for a pointer is needed

A diagram of a stack

Description automatically generated

* Before popping the element from the stack, we check if the stack is **empty**.
* If the stack is empty (head == nul), then **Stack Underflows**and we cannot remove any element from the stack.
* Otherwise, we store the value at top, decrement the value of top by 1 **(top = top – 1)**and return the stored top value.

bool Stack::pop()

{

    if (isEmpty())

    {

        return false; // Stack is empty

    }

    else

    {

        Node \*temp = topNode;    // Save the top node

        topNode = topNode->next; // Update topNode to the next node

        delete temp;             // Delete the top node

        return true;

    }

}

bool Stack::pop(ItemType &item)

{

    if (isEmpty())

    {

        return false; // Stack is empty

    }

    else

    {

        item = topNode->item; // Retrieve the item

        return pop();         // Pop the top node

    }

}

## getTop(ItemType &item) – Returns top element from stack:

Returns top element from stack

Time Complexity: O(1) - Constant time as it only accesses the top element

Space Complexity: O(1) - No extra space required

A diagram of a stack

Description automatically generated

* Before returning the top element from the stack, we check if the stack is empty.
* If the stack is empty (head == nullptr), we simply print “Stack is empty”.
* Otherwise, we return the element stored at **index = top**.

void Stack::getTop(ItemType &item)

{

    if (isEmpty())

    {

        cout << "Stack is empty." << endl;

    }

    else

    {

        item = topNode->item;

    }

}

## isEmpty() – Returns true if empty, else false

Returns boolean, true/false if empty

Time Complexity: O(1)

 Space Complexity: O(1)

A diagram of a stack

Description automatically generated

* Check if head == nullptr

    bool isEmpty() const {

        return root == nullptr;

    }

## append() – Adds to the end of the stack

Add an item to the back of the stack (treated as a list)

Time Complexity: O(n) - Linear time as it may need to traverse the entire stack to find the end.

Space Complexity: O(1) - Only a single pointer is allocated for the new node.

* Create a newNode
* Check if stack is empty for fringe case
* Intialise “current” pointer to root
* Traverse to the end of the stack, set pointer of last node to point to newNode

    void append(int data) {

        StackNode\* newNode = new StackNode(data);  // Create a new node with the specified data

        if (isEmpty()) {                            // If the stack is empty, the new node becomes the root

            root = newNode;

            cout << data << " appended to stack\n";

            return;

        }

        StackNode\* current = root;                  // Initialize 'current' to start at the root of the stack

        // Traverse to the last node in the stack

        while (current->next != nullptr) {

            current = current->next;                // Move to the next node

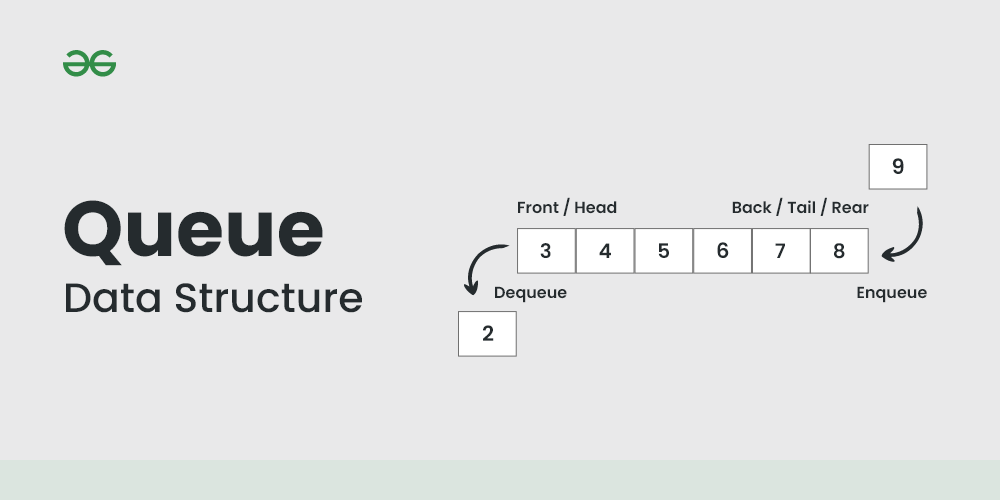
        }

        current->next = newNode;                     // Link the last node to the new node

        cout << data << " appended to stack\n";       // Output the appended element

    }

# Queues:



Queue is “First in, first out”, where the first element added to the queue si the first one to be removed.

**Basic Operations on Queue Data Structure (Pointers only):**

To make manipulations in a stack, there are certain operations provided to us.

1. **Constructor** to create an empty queue
2. **enqueue(ItemType& item)** insertion of elements to the queue
3. **dequeue()** removal of elements from the queue
4. **dequeue((ItemType& item)** removal of element from queue and returned in function
5. **getFront()**acquires the data element available at the front node of the queue without
6. **isEmpty()**returns true if queue is empty
7. **size()** returns the size of the queue
8. **Destructor** to destroy a queue

## Constructor:

template <typename T>

Queue<T>::Queue() {

    frontNode = nullptr;

    backNode = nullptr;

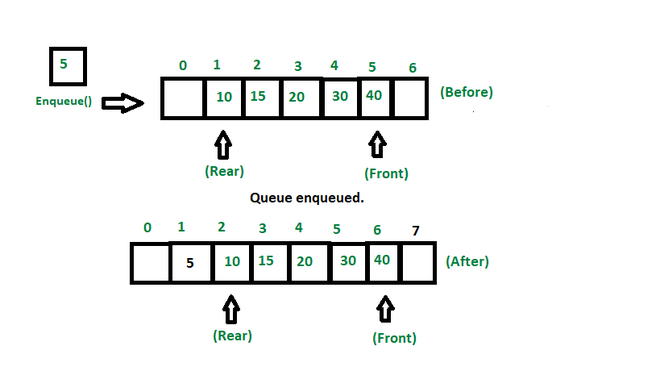
}

## Enqueue(ItemType &item):

Adds an item to back of the queue

Time Complexity: O(1) because insertion happens at the end

Space Complexity: O(1) per operation, but O(n) for n items



bool Queue::enqueue(ItemType& item)

{

    Node\* newNode = new Node;

    if (newNode == nullptr) // Check if memory allocation failed

    {

        return false;

    }

    newNode->item = item;

    newNode->next = nullptr;

    if (isEmpty())

    {

        frontNode = newNode;

    }

    else

    {

        backNode->next = newNode;

    }

    backNode = newNode;

    return true;

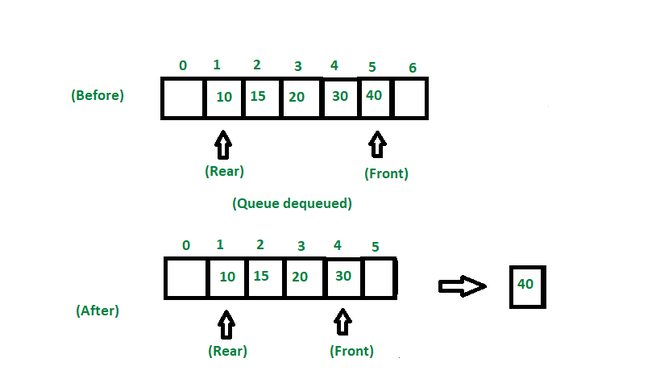
}

## Dequeue():

Removes an item at the front of the queue

Time Complexity: O(1), removed at front node

Space Complexity: O(1)



// Dequeue an item from the queue (without returning the item)

bool Queue::dequeue()

{

    if (isEmpty())

    {

        return false;

    }

    Node\* tempNode = frontNode;

    frontNode = frontNode->next;

    if (frontNode == nullptr) // Queue is now empty

    {

        backNode = nullptr;

    }

    delete tempNode;

    return true;

}

## Dequeue(ItemType& Item):

Removes an item at the front of the queue, but parses the item by reference

Time Complexity: O(1)

Space Complexity: O(1)

// Dequeue an item from the queue (and return the item by reference)

bool Queue::dequeue(ItemType& item)

{

    if (isEmpty())

    {

        return false;

    }

    item = frontNode->item;

    return dequeue();

}

## getFront(ItemType& Item):

Retrieves the front item in the queue without dequeuing it

Time Complexity: O(1)

Space Complexity: O(1)

// Get the front item of the queue without dequeuing it

void Queue::getFront(ItemType& item)

{

    if (!isEmpty())

    {

        item = frontNode->item;

    }

    else

    {

        cout << "Queue is empty." << endl;

    }

}

## isEmpty():

Checks if the queue is empty

Time Complexity: O(1)

Space Complexity: O(1)

bool Queue::isEmpty()

{

    return frontNode == nullptr;

}

## Destructor:

template <typename T>

Queue<T>::~Queue() {

    while (!isEmpty()) {

        dequeue();

    }

}

# Hash Tables:

A screenshot of a math equation

Description automatically generated

Hashing is a technique used in data structures that efficiently stores and retrieves data in a way that allows for quick access

* Involves mapping data to a specific index in a hash table using a hash function that enables fast retrieval of information based on its key
* O(1) for search, insert and delete on average
* Used to implement a set of distinct items and dictionaries (key value pairs)

Separate chaining:

Each array element is a linked list, when collision occurs, item is added to list

**Basic Operations on Hash Table Data Structure:**

To make manipulations in a hash table, there are certain operations provided to us.

1. **Constructor** to create an empty hash table
2. **Add(KeyType, ItemType): bool** to add an item to a specific key
3. **Remove(KeyType): void** to remove an item at a specific key
4. Get(KeyType): void to retrieve an item at a specific key
5. **isEmpty(): bool** to validate if hash table is empty
6. **getLength(): int** to check length of hash table
7. **Destructor** to destroy an empty hash table

## Constructor:

Dictionary::Dictionary() : size(0)

{

    for (int i = 0; i < MAX\_SIZE; ++i)

    {

        items[i] = nullptr;

    }

}

## Add(KeyType, ItemType):

Adds an item to the hash table

Time Complexity: [This depends on hash function]

Average Case: O(1)

Worst Case: O(n)

Space Complexity: O(1) per operation, O(n) total

// add a new item with the specified key to the Dictionary

bool Dictionary::add(KeyType newKey, ItemType newItem)

{

    // get the index of the new key

    int index = hash(newKey);

    // create a new node

    Node \*newNode = new Node;

    // assign the key and item to the new node

    newNode->key = newKey;

    // assign the item to the new node

    newNode->item = newItem;

    // assign the next pointer to the new node

    newNode->next = nullptr;

    // if the index is empty

    if (items[index] == nullptr)

    {

        // assign the new node to the index

        items[index] = newNode;

        size++;

        return true;

    }

    else

    {

        // Create a pointer to the current node residing in the index

        Node \*current = items[index];

        // Traverse the linked list until the end

        while (current != nullptr)

        {

            // if the key already exists

            if (current->key == newKey)

            {

                // delete the new node, clear the memory

                delete newNode;

                return false;

            }

            // if at the end of list, attach new node

            if (current->next == nullptr)

            {

                current->next = newNode;

                size++;

                return true;

            }

            // move to the next node

            current = current->next;

        }

    }

    // If failed delete node in case of memory leak

    delete newNode;

    return false;

}

## Remove(KeyType):

Removes an item from the hash table by key

Time Complexity: [This depends on hash function]

Average Case: O(1)

Worst Case: O(n)

Space Complexity: O(1) per operation, O(n) total

// remove an item with the specified key in the Dictionary

void Dictionary::remove(KeyType key)

{

    // get the index of the key

    int index = hash(key);

    // Get the current node

    Node \*current = items[index];

    // Get the previous node

    Node \*previous = nullptr;

    // Traverse the linked list

    while (current != nullptr)

    {

        // If statement to check if the key is found

        if (current->key == key)

        {

            // If the previous node is null, check if it is the first node

            if (previous == nullptr)

            {

                // If it is the first node, assign the next node to index

                items[index] = current->next;

            }

            else

            {

                // If it is not the first node, assign the next node to the previous node

                // This changes the pointer of next for previous node, to be the same pointer of next for current node (skips over current node)

                previous->next = current->next;

            }

            delete current;

            size--;

            return;

        }

        // Move to the next node

        previous = current;

        current = current->next;

    }

    cerr << "Key not found" << endl;

}

## get(KeyType):

Return item, from a specific key value

Time Complexity:

Average Case: O(1)

Worst Case: O(n)

Space Complexity: O(1)

ItemType Dictionary::get(KeyType key)

{

    // get the index of the key

    int index = hash(key);

    // Get the current node

    Node \*current = items[index];

    // Traverse the linked list

    while (current != nullptr)

    {

        // If statement to check if the key is found

        if (current->key == key)

        {

            return current->item;

        }

        else

        {

            // Move to the next node

            current = current->next;

        }

    }

    throw "Key not found";

}

## isEmpty():

Time Complexity: O(1)

Checking if the hash table is empty just involves comparing the size to 0.

Space Complexity: O(1)

// check if the Dictionary is empty

bool Dictionary::isEmpty()

{

    return size == 0;

}

## getLength()

Time Complexity: O(1)

Simply returns the current size of the hash table, which is usually stored as an internal

variable. Space Complexity: O(1)

// check the size of the Dictionary

int Dictionary::getLength()

{

    return size;

}

## Destructor()

Time Complexity: O(n)

Involves deallocating the memory used by the hash table, which involves visiting every element to free it.

Space Complexity: O(1) (since it frees memory and doesn’t require extra space other than traversal)

// Destructor

Dictionary::~Dictionary()

{

    for (int i = 0; i < MAX\_SIZE; ++i)

    {

        Node \*current = items[i];

        while (current != nullptr)

        {

            Node \*temp = current;

            current = current->next;

            delete temp;

        }

    }

}

## Binary Search Trees

* Tree is a data structure that organises data in a hierarchical order

Root: Top node of the tree

Leaf: Node with no children

Level: Number of generations from root

Height: Number of levels in the tree

Parent: Node with nodes (children) below it

Children: Nodes below a given node (parent)

General Tree:

* Tree with any number of subtrees

Binary Tree:

* A tree with at most 2 subtrees

Binary Search Tree:

* A binary tree that is ordered:
  + Values on left subtree < value of parent
  + Values on right subtree > value of parent

AVL Tree

* A binary search tree that is balanced
  + Heights of any node’s two subtrees differ by at most 1

Structure of a Node in Binary Tree:

struct BinaryNode

{

   ItemType item;      // data item

   BinaryNode\* left;   // pointer pointing to left subtree

   BinaryNode\* right;  // pointer pointing to right subtree

};

Full Binary Trees:

* Tree is full if every node (except the leaf nodes) has two children
* Number of nodes in a full binary tree of height h, n = 2^h – 1
* Height of a full binary tree with a n nodes is h = log2(n+1)

A diagram of a network

Description automatically generated

Complete Binary Trees:

* Full to all levels except the last level
* Last level filled from left to right

A blue line drawing of a network

Description automatically generated

Balanced Binary Trees:

* Balanced if the heights of any node’s two subtrees differ by at most 1

A close-up of a diagram

Description automatically generated

## Traversal of Binary Tree

1. In order – Left-Root-Right
2. Preorder – Root-Left-Right
3. Post order – Left-Right-Root
4. Level order – Level by Level

A diagram of a network

Description automatically generated

### In-Order Traversal

1. Calls root->left, does all left subtree nodes is ascending order
2. Process current node by printing root->data
3. Recurses on root->right, does all right subtree nodes in ascending order

1→ 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11

void inorderTraversal(TreeNode\* root) {

    if (root == nullptr) {

        return;

    }

    // 1) Traverse left subtree

    inorderTraversal(root->left);

    // 2) Visit current node

    cout << root->data << " ";

    // 3) Traverse right subtree

    inorderTraversal(root->right);

}

### Pre-Order Traversal

1. Visits root node
2. Traverse left subtree recursively
3. Traverse right subtree recursively

6 → 4 → 2 → 1 → 3 → 5 → 8 → 7 → 10 → 9 → 11

void preorderTraversal(TreeNode\* root) {

    if (root == nullptr) {

        return;

    }

    // Visit the root node

    cout << root->data << " ";

    // Traverse the left subtree

    preorderTraversal(root->left);

    // Traverse the right subtree

    preorderTraversal(root->right);

}

### Post Order Traversal

1. Traverse left subtree recursively
2. Traverse right subtree recursively
3. Visit root node

1 3 2 5 4 7 9 11 10 8 6

1 → 3 → 2 → 5 → 4 → 7 → 9 → 11 → 10 → 8 → 6

void postorderTraversal(TreeNode\* root) {

    if (root == nullptr) {

        return;

    }

    // Traverse the left subtree

    postorderTraversal(root->left);

    // Traverse the right subtree

    postorderTraversal(root->right);

    // Visit the root node

    cout << root->data << " ";

}

### Level Order Traversal

Traverse level by level, start with parent, left child, right child

6 → 4 → 8 → 2 → 5 → 7 → 10 → 1 → 3 → 9 → 11

void levelOrderTraversal(TreeNode\* root) {

    if (root == nullptr) {

        return;

    }

    // Use a queue to hold nodes at each level

    queue<TreeNode\*> q;

    q.push(root);

    while (!q.empty()) {

        TreeNode\* current = q.front();

        q.pop();

        // Print the current node's data

        cout << current->data << " ";

        // Enqueue left child if it exists

        if (current->left != nullptr) {

            q.push(current->left);

        }

        // Enqueue right child if it exists

        if (current->right != nullptr) {

            q.push(current->right);

        }

    }

}

## Operations of Binary Search Tree:

### Search:

A diagram of a network

Description automatically generated



1. Starts at root node
2. Compare key with node values, if smaller traverse left subtree, if larger traverse right subtree
3. Repeat until key found or nullptr encountered

* Time Complexity: O(logn) if balanced, O(n) if skewed
* Space Complexity: O(h) where h is height of tree, due to recursion stack

bool searchBST(TreeNode\* root, int key) {

    if (root == nullptr) {

        return false; // Key not found

    }

    if (root->data == key) {

        return true; // Key found

    }

    if (key < root->data) {

        return searchBST(root->left, key); // Search left subtree

    } else {

        return searchBST(root->right, key); // Search right subtree

    }

}

### Insert

A diagram of a diagram

Description automatically generated



1. Starts from root node
2. Compare key to be inserted with current node:
   1. If key is less than current node, move to left subtree
   2. If key is more than current node, move to right subtree
3. Repeat process recursively until empty node (nullptr) is found
4. Insert new node at empty spot

Time Complexity: O(logn) for balanced BSTs, O(n) for skewed BSTs

Space Complexity: O(h) where h is the height of the tree (due to recursion stack)

// Insert Function in BST

TreeNode\* insertBST(TreeNode\* root, int key) {

    // If the tree is empty, create a new node

    if (root == nullptr) {

        return new TreeNode(key);

    }

    // If the key is smaller, go to the left subtree

    if (key < root->data) {

        root->left = insertBST(root->left, key);

    }

    // If the key is larger, go to the right subtree

    else if (key > root->data) {

        root->right = insertBST(root->right, key);

    }

    // Return the unchanged root node

    return root;

}

### Deletion

Case 1: Leaf Node  
A diagram of a tree

Description automatically generated

Case 2: Delete node w/ one child

A diagram of a tree

Description automatically generated

Case 3: Delete node with both children in BST

A diagram of a diagram

Description automatically generated

Note: Utilize in order successor

// Find the Inorder Successor (Smallest in Right Subtree)

TreeNode\* findMin(TreeNode\* node) {

    while (node && node->left != nullptr) {

        node = node->left;

    }

    return node;

}

// Delete Node in BST

TreeNode\* deleteBST(TreeNode\* root, int key) {

    if (root == nullptr) {

        return nullptr;

    }

    // Search for the node

    if (key < root->data) {

        root->left = deleteBST(root->left, key);

    } else if (key > root->data) {

        root->right = deleteBST(root->right, key);

    } else {

        // \*\*Case 1: Leaf Node\*\*

        if (root->left == nullptr && root->right == nullptr) {

            delete root;

            return nullptr;

        }

        // \*\*Case 2: One Child\*\*

        if (root->left == nullptr) {

            TreeNode\* temp = root->right;

            delete root;

            return temp;

        } else if (root->right == nullptr) {

            TreeNode\* temp = root->left;

            delete root;

            return temp;

        }

        // \*\*Case 3: Two Children\*\*

        TreeNode\* temp = findMin(root->right); // Find Inorder Successor

        root->data = temp->data; // Copy value

        root->right = deleteBST(root->right, temp->data); // Delete successor

    }

    return root;

}

### Minimum Value in BST:

1. Start at root node
2. Null check for nullptr, prevents undefined behaviour
3. Uses pointer “current” to traverse tree, move left repeatedly
4. Return left most node

Time Complexity: O(h) which is height of tree

Space Complexity: O(1) iterative approach

// Find the Inorder Successor (Smallest node in the right subtree)

TreeNode\* findMin(TreeNode\* node) {

    if (node == nullptr) {

        return nullptr; // Empty subtree

    }

    TreeNode\* current = node;

    while (current->left != nullptr) {

        current = current->left; // Keep moving left

    }

    return current; // Return the leftmost node

}

### Maximum Value in BST:

1. Start at root node
2. Null check for nullptr, prevents undefined behaviour
3. Uses pointer “current” to traverse tree, move right repeatedly
4. Return right most node

Time Complexity: O(h), where h is the height of the tree

Space Complexity: O(1), iterative approach

TreeNode\* findMax(TreeNode\* node) {

    if (node == nullptr) {

        return nullptr; // Empty subtree

    }

    TreeNode\* current = node;

    while (current->right != nullptr) {

        current = current->right; // Keep moving right

    }

    return current; // Return the rightmost node

}

### Floor:

1. Start at root node
2. If the current node's value is equal to xxx → return the current node.
3. If the current node's value is greater than xxx → move to the left subtree.
4. If the current node's value is less than xxx:
   1. Store this node as a candidate (potential floor value).
   2. Move to the right subtree to check for a larger valid floor value.
5. Repeat until traversal ends

Time Complexity: O(h) where h is the height of the BST

Space Complexity: O(1), iterative approach

A diagram of numbers and points

Description automatically generated

TreeNode\* findFloor(TreeNode\* root, int key) {

    TreeNode\* floorNode = nullptr;

    while (root != nullptr) {

        if (root->data == key) {

            // Exact match, this is the floor

            return root;

        }

        else if (key < root->data) {

            // Move to the left subtree

            root = root->left;

        }

        else {

            // Store the current node as a potential floor

            floorNode = root;

            // Move to the right subtree

            root = root->right;

        }

    }

    return floorNode;

}

### Ceiling:

1. Start from the root node.
2. Compare the current node's value with xxx:
   1. If the current node's value is equal to xxx → return the current node.
   2. If the current node's value is smaller than xxx → move to the right subtree.
3. If the current node's value is greater than xxx:
   1. Store this node as a candidate (potential ceiling value).
   2. Move to the left subtree to check for a smaller valid ceiling value.
4. Repeat until you reach the end of the tree.

Time Complexity: O(h), where h is height of the BST

Space Complexity: O(1), iterative approach

// Find Ceiling in BST

TreeNode\* findCeiling(TreeNode\* root, int key) {

    TreeNode\* ceilingNode = nullptr;

    while (root != nullptr) {

        if (root->data == key) {

            // Exact match, this is the ceiling

            return root;

        }

        else if (key > root->data) {

            // Move to the right subtree

            root = root->right;

        }

        else {

            // Store the current node as a potential ceiling

            ceilingNode = root;

            // Move to the left subtree

            root = root->left;

        }

    }

    return ceilingNode;

}

### In order predecessor:

1. If the node has a left subtree
   1. In order predecessor is the rightmost node in left subtree
2. If node does not have a left subtree
   1. Start from root
   2. Use a successor pointer to keep track of valid predecessor
   3. Move right if node’s value is smaller than current node
   4. Move left if the node’s value is greater than the current node

Time Complexity: O(h) where h is the height of the BST

Space Complexity: O(1) iterative traversal

Needs to use findMax

// Find Inorder Predecessor in BST

TreeNode\* findPredecessor(TreeNode\* root, TreeNode\* target) {

    if (target == nullptr) return nullptr;

    // Case 1: Node has a left subtree

    if (target->left != nullptr) {

        return findMax(target->left);

    }

    // Case 2: Node does not have a left subtree

    TreeNode\* predecessor = nullptr;

    TreeNode\* current = root;

    while (current != nullptr) {

        if (target->data > current->data) {

            predecessor = current; // Update predecessor

            current = current->right;

        } else if (target->data < current->data) {

            current = current->left;

        } else {

            break;

        }

    }

    return predecessor;

}

### In Order Successor:

1. If the node has a right subtree
   1. The in-order successor is the leftmost (minimum) node in the right subtree
2. If the node does not have a right subtree
   1. Start from the root
   2. Use a successor pointer to keep track of a valid successor
   3. Move left if the node’s value is smaller than the current node’s value
   4. Move right if the node’s value is greater than the current node's value

Time Complexity: O(h) where h is the height of the BST

Space Complexity: O(1) iterative traversal

Needs to use findMin

// Find Inorder Successor in BST

TreeNode\* findSuccessor(TreeNode\* root, TreeNode\* target) {

    if (target == nullptr) return nullptr;

    // Case 1: Node has a right subtree

    if (target->right != nullptr) {

        return findMin(target->right);

    }

    // Case 2: Node does not have a right subtree

    TreeNode\* successor = nullptr;

    TreeNode\* current = root;

    while (current != nullptr) {

        if (target->data < current->data) {

            successor = current; // Update successor

            current = current->left;

        } else if (target->data > current->data) {

            current = current->right;

        } else {

            break;

        }

    }

    return successor;

}

## Handling Duplicates:

* Update TreeNode to store a count variable
* Update Insertion, Search and Deletion functions

struct TreeNode {

    int data;

    int count; // Count occurrences of the same value

    TreeNode\* left;

    TreeNode\* right;

    TreeNode(int val) : data(val), count(1), left(nullptr), right(nullptr) {}

};

Insertion:

* If the value already exists in the tree, increment its count.
* Otherwise, proceed as usual to insert a new node.

// --- Insertion with Duplicate Handling ---

TreeNode\* insertBST(TreeNode\* root, int key) {

    if (root == nullptr) return new TreeNode(key);

    if (key == root->data) {

        root->count++; // Increment count for duplicates

    } else if (key < root->data) {

        root->left = insertBST(root->left, key);

    } else {

        root->right = insertBST(root->right, key);

    }

    return root;

}

Search:

* Check if the node with the given value exists but do nothing with the count.

// --- Search ---

bool searchBST(TreeNode\* root, int key) {

    if (root == nullptr) return false;

    if (key == root->data) return true;

    else if (key < root->data) return searchBST(root->left, key);

    else return searchBST(root->right, key);

}

Deletion:

* If the node has a count > 1, decrement the count instead of removing the node.
* If the count == 1, remove the node following the standard deletion rules.

// --- Deletion with Duplicate Handling ---

TreeNode\* deleteBST(TreeNode\* root, int key) {

    if (root == nullptr) return nullptr;

    if (key < root->data) {

        root->left = deleteBST(root->left, key);

    } else if (key > root->data) {

        root->right = deleteBST(root->right, key);

    } else {

        // Handle duplicate count

        if (root->count > 1) {

            root->count--;

            return root;

        }

        // Case 1: Leaf Node

        if (root->left == nullptr && root->right == nullptr) {

            delete root;

            return nullptr;

        }

        // Case 2: One Child

        if (root->left == nullptr) {

            TreeNode\* temp = root->right;

            delete root;

            return temp;

        } else if (root->right == nullptr) {

            TreeNode\* temp = root->left;

            delete root;

            return temp;

        }

        // Case 3: Two Children

        TreeNode\* temp = findMin(root->right);

        root->data = temp->data;

        root->count = temp->count;

        temp->count = 1;

        root->right = deleteBST(root->right, temp->data);

    }

    return root;

}

# AVLs

* Binary search tree
* Is always balanced
* Will re-balance itself when it becomes unbalanced
* No. of comparisons worse case = O(log2n)

|  |  |  |
| --- | --- | --- |
| Tree Condition | Subtree Condition | Rotation Type |
| Right Heavy | Right Subtree is NOT left heavy | Left Rotation |
| Right Heavy | Right Subtree is left heavy | Right-Left Rotation |
| Left Heavy | Left Subtree is NOT right heavy | Right Rotation |
| Left Heavy | Left Subtree is right heavy | Left-Right Rotation |

**Left Rotation:** Performed when a node becomes unbalanced due to an insertion in the right subtree of the right child.

A diagram of a tree rotation

Description automatically generated

**Right Rotation:** Performed when a node becomes unbalanced due to an insertion in the left subtree of the left child.

A diagram of a tree rotation

Description automatically generated

**Right-Left Rotation:** A combination of Right Rotation on the right child followed by a Left Rotation.

A diagram of a right-left rotation

Description automatically generated

**Left-Right Rotation:** A combination of Left Rotation on the left child followed by a Right Rotation

A diagram of a left-right rotation

Description automatically generated

Advantages:

1. AVL trees can self-balance themselves and therefore provides time complexity as O (Log n) for search, insert and delete.
2. It is a BST only (with balancing), so items can be traversed in sorted order.
3. Since the balancing rules are strict compared to Red Black Tree, AVL trees in general have relatively less height and hence the search is faster.
4. AVL tree is relatively less complex to understand and implement compared to Red Black Trees.

Disadvantages:

1. It is difficult to implement compared to normal BST and easier compared to Red Black
2. Less used compared to Red-Black trees. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.

## Insertion:

Let newly inserted node be W

1. Perform standard BST insert for W
2. Starting from W, travel up to find the first unbalanced node, let z be the first unbalanced node, y be the child of z that comes on the path from w to z, and x be the grandchild of z that comes on the path from w to z
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z, there can be 4 possible cases
4. Following are the possible 4 arrangements:

* y is the left child of z and x is the left child of y (Left Left Case)
* y is the left child of z and x is the right child of y (Left Right Case)
* y is the right child of z and x is the right child of y (Right Right Case)
* y is the right child of z and x is the left child of y (Right Left Case)

A screenshot of a computer

Description automatically generated

Left-Left Case:  
A triangle with letters and numbers

Description automatically generated

Left-Right Case:

A close-up of a computer screen

Description automatically generated

Right-Right Case:  
A bird flying in the sky

Description automatically generated

Right-Left Case:

A white background with black text

Description automatically generated

### Deletion:

Let w be the node to be deleted

1. Perform standard BST delete for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here.
3. Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
   1. y is left child of z and x is left child of y (Left Left Case)
   2. y is left child of z and x is right child of y (Left Right Case)
   3. y is right child of z and x is right child of y (Right Right Case)
   4. y is right child of z and x is left child of y (Right Left Case)

A screenshot of a computer program

Description automatically generated

Left Left Case:  
A triangle with numbers and letters

Description automatically generated with medium confidence

Left Right Case:

A screen shot of a computer

Description automatically generated

Right Right Case:

A computer screen shot of a person

Description automatically generated

Right Left Case:

A white background with black text

Description automatically generated

Code:  
Utility Function to getHeight

// Utility Function: Get the height of a node

int height(Node\* node) {

    return (node == nullptr) ? 0 : node->height;

}

### Utility Function to getBalance

int getBalance(Node\* node) {

    return (node == nullptr) ? 0 : height(node->left) - height(node->right);

}

### rotateRight Function

// Utility Function: Perform Right Rotation

Node\* rightRotate(Node\* y) {

    Node\* x = y->left;

    Node\* T2 = x->right;

    // Rotation

    x->right = y;

    y->left = T2;

    // Update heights

    y->height = max(height(y->left), height(y->right)) + 1;

    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root

    return x;

}

### rotateLeft Function

Node\* leftRotate(Node\* x) {

    Node\* y = x->right;

    Node\* T2 = y->left;

    // Rotation

    y->left = x;

    x->right = T2;

    // Update heights

    x->height = max(height(x->left), height(x->right)) + 1;

    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root

    return y;

}

### Insert Function:

Node\* insert(Node\* node, int key) {

    // 1. Perform standard BST insertion

    if (node == nullptr)

        return new Node(key);

    if (key < node->key)

        node->left = insert(node->left, key);

    else if (key > node->key)

        node->right = insert(node->right, key);

    else

        return node; // Duplicate keys are not allowed

    // 2. Update height of the current node

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

    // 3. Get the balance factor

    int balance = getBalance(node);

    // 4. Handle Imbalance Cases

    // \*\*Left-Left (LL) Case\*\*

    if (balance > 1 && key < node->left->key)

        return rightRotate(node);

    // \*\*Right-Right (RR) Case\*\*

    if (balance < -1 && key > node->right->key)

        return leftRotate(node);

    // \*\*Left-Right (LR) Case\*\*

    if (balance > 1 && key > node->left->key) {

        node->left = leftRotate(node->left);

        return rightRotate(node);

    }

    // \*\*Right-Left (RL) Case\*\*

    if (balance < -1 && key < node->right->key) {

        node->right = rightRotate(node->right);

        return leftRotate(node);

    }

    // Return the unchanged node pointer

    return node;

}

### Deletion:

Node\* deleteNode(Node\* root, int key) {

    // Step 1: Perform standard BST deletion

    if (root == nullptr)

        return root;

    if (key < root->key)

        root->left = deleteNode(root->left, key);

    else if (key > root->key)

        root->right = deleteNode(root->right, key);

    else {

        // Node with only one child or no child

        if (root->left == nullptr || root->right == nullptr) {

            Node\* temp = root->left ? root->left : root->right;

            if (temp == nullptr) {

                temp = root;

                root = nullptr;

            } else {

                \*root = \*temp; // Copy the contents of the non-empty child

            }

            delete temp;

        } else {

            // Node with two children: Get inorder successor (smallest in right subtree)

            Node\* temp = findMin(root->right);

            root->key = temp->key;

            root->right = deleteNode(root->right, temp->key);

        }

    }

    if (root == nullptr)

        return root;

    // Step 2: Update height

    root->height = 1 + max(height(root->left), height(root->right));

    // Step 3: Check balance and perform rotations if necessary

    int balance = getBalance(root);

    // \*\*Left-Left (LL) Case\*\*

    if (balance > 1 && getBalance(root->left) >= 0)

        return rightRotate(root);

    // \*\*Left-Right (LR) Case\*\*

    if (balance > 1 && getBalance(root->left) < 0) {

        root->left = leftRotate(root->left);

        return rightRotate(root);

    }

    // \*\*Right-Right (RR) Case\*\*

    if (balance < -1 && getBalance(root->right) <= 0)

        return leftRotate(root);

    // \*\*Right-Left (RL) Case\*\*

    if (balance < -1 && getBalance(root->right) > 0) {

        root->right = rightRotate(root->right);

        return leftRotate(root);

    }

    return root;

}

## Sorting Techniques

* + Process that organises a collection of data into either ascending or descending order
  + Sort Key part of the data item that we would consider during sorting

### Selection Sort

A diagram of a number

Description automatically generated

A diagram of a number

Description automatically generated

* + Comparison based sorting algorithm – sorts array by repeatedly selecting the smallest element from unsorted portion and swapping it with the first unsorted element -> Continues until the entire array is sorted

1. First find the smallest element and swap it with the first element. (Get the smallest element at its correct position)
2. Find the smallest among remaining elements and swap it with the second element
3. Continue until all elements in correct position

void selectionSort(vector<int> &arr) {

    int n = arr.size();

    for (int i = 0; i < n - 1; ++i) {

        // Assume the current position holds

        // the minimum element

        int min\_idx = i;

        // Iterate through the unsorted portion

        // to find the actual minimum

        for (int j = i + 1; j < n; ++j) {

            if (arr[j] < arr[min\_idx]) {

                // Update min\_idx if a smaller

                // element is found

                min\_idx = j;

            }

        }

        // Move minimum element to its

        // correct position

        swap(arr[i], arr[min\_idx]);

    }

}

Time Complexity: O(N^2) because of nested loops, inefficient for large datasets

Space Complexity: O(1) since it sorts array in place

Does not depend on initial arrangement of data, only appropriate for small n

## Insertion Sort

A screenshot of a computer

Description automatically generated

* + Simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted position of the list

1. Partition the array into two regions: sorted and unsorted
2. Take each item from the unsorted region and insert it into its correct order in the sorted region

void insertionSort(int arr[], int n)

{

    for (int i = 1; i < n; ++i) {

        // Store the value of the current element in a temp variable

        int key = arr[i];

        // Initialize j to be one index behind i

        int j = i - 1;

        /\* Move elements of arr[0..i-1], that are

           greater than key, to one position ahead

           of their current position \*/

        while (j >= 0 && arr[j] > key) {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

Time Complexity: Worst Case – O(n^2), Best Case – O(n) [Nested for loop]

Space Complexity – O(1) since it sorts the array in place

Appropriate for small arrays due to simplicity, prohibitively inefficient for large arrays

Strengths:

* Good when unordered list is mostly sorted
* Need minimum time to verify if list is sorted
* Better w/ pointer-based implementation

Weaknesses:

* Every new insertion requires movements/shifting for some inserted items in ordered position
* Each slot contains large record => movement is expensive
* Array based implementation less suitable

## Merge Sort

A diagram of a large group

Description automatically generated with medium confidence

* + Sorting algorithm that follows divide-and-conquer approach. Works recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array

1. Divide: The list or array recursively into two halves until it can no more be divided
2. Conquer: Each subarray is sorted individually using the merge sort algorithm
3. Merge: Sorted subarrays are merged back together in sorted order; process continues until all elements from both subarrays have been merged

void merge(int array[], int leftIndex, int midIndex, int rightIndex) {

        int leftArraySize = midIndex - leftIndex + 1;

        int rightArraySize = rightIndex - midIndex;

        // Create two temporary subarrays

        int leftArray[leftArraySize];

        int rightArray[rightArraySize];

        // Copy the elements of the original array into the two subarrays

        for (int i = 0; i < leftArraySize; i++)

            leftArray[i] = array[leftIndex + i];

        for (int j = 0; j < rightArraySize; j++)

            rightArray[j] = array[midIndex + 1 + j];

        // Merge the two subarrays back into the original array

        int index = leftIndex;

        int i = 0;

        int j = 0;

        while (i < leftArraySize && j < rightArraySize) {

            if (leftArray[i] <= rightArray[j]) {

                array[index] = leftArray[i];

                index++;

                i++;

            }

            else {

                array[index] = rightArray[j];

                index++;

                j++;

            }

        }

        // Copy the remaining elements of the left subarray into the original array

        while (i < leftArraySize) {

            array[index] = leftArray[i];

            index++;

            i++;

        }

        // Copy the remaining elements of the right subarray into the original array

        while (j < rightArraySize) {

            array[index] = rightArray[j];

            index++;

            j++;

        }

    }

void mergeSort(int array[], int leftIndex, int rightIndex) {

    if (leftIndex >= rightIndex) return;

    // Find the middle index

    int midIndex = leftIndex + (rightIndex - leftIndex) / 2;

    // Recursively sort left and right halves of the array

    mergeSort(array, leftIndex, midIndex);

    mergeSort(array, midIndex+1, rightIndex);

    // Merge the two sorted halves of the array

    merge(array, leftIndex, midIndex, rightIndex);

}

Time Complexity: O (n log n), when array is already sorted or nearly sorted

Space Complexity: O (n), additional space required for temporary array used during merging

* Performance is independent of initial order of the array items

Strengths:

* Extremely fast algorithm, good runtime behaviour
* Implements easily when using pointer-based implementation

Weaknesses:

* For array implementation, need auxiliary storage -> difficult to implement for array implementation

## Quick Sort

* + Fastest general purpose in-memory sorting algorithm in the average case

A screenshot of a diagram

Description automatically generated

Divide and conquer, breaking down the problem into smaller sub-problems

1. Choose a pivot: Select an element from the array as the pivot. Choice of pivot can vary (first element, last element, random element, or median)
2. Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain index of the pivot
3. Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot)
4. Base Case: Recursion stops when there is only one element left in the sub-array, as a single element is already sorted

A diagram of a pivot

Description automatically generated

Choices for picking pivots:

* Always pick the first (or last) element as a pivot. Ends up as worst-case scenario if array already sorted
* Pick random element as pivot. Preferred approach as it does not have a pattern for worst case
* Pick the median element is pivot. Ideal approach in terms of time complexity, find median in linear time and partition function will always divide the input array into two halves -> but more time on average

Partition Algorithm

1. Naïve Partition – Create a copy of the array. First put all smaller elements and then all greater. Finally, we copy the temporary array back to original array. This requires O(n) extra space.
2. Lomuto Partition – Have used this partition in this article. This simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity
3. Hoare’s Partition – This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned.

Time Complexity – O (n log n), occurs when the pivot element divides the array into two equal halves

Space Complexity – O(n), due to recursive call stack

int partition(vector<int>& arr, int low, int high) {

    // Choose the pivot

    int pivot = arr[high];

    // Index of smaller element and indicates

    // the right position of pivot found so far

    int i = low - 1;

    // Traverse arr[;ow..high] and move all smaller

    // elements on left side. Elements from low to

    // i are smaller after every iteration

    for (int j = low; j <= high - 1; j++) {

        if (arr[j] < pivot) {

            i++;

            swap(arr[i], arr[j]);

        }

    }

    // Move pivot after smaller elements and

    // return its position

    swap(arr[i + 1], arr[high]);

    return i + 1;

}

// The QuickSort function implementation

void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        // pi is the partition return index of pivot

        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements

        // and greater or equals elements

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

Strengths:

* Fast on average
* No merging required
* Best case if pivot always splits data into equal halves

Weaknesses:

* Pivots need to be chosen carefully
* Quicksort performs badly when the array is already sorted, and pivot is largest or smallest element
* Performs badly when n is small

## Radix Sort

* + Treats each data element as a character string
  + Repeatedly organises the data into groups according to the ith character/digit in each element

A screenshot of a number

Description automatically generated

1. Find the largest element in the array, which is 802. Has 3 digits, will iterate three times, once for each significant place

A screenshot of a computer

Description automatically generated

1. Sort based on the unit place digits

Sort the elements based on the unit place digits (X = 0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place. It’s important to understand that the default implementation of counting sort is unstable i.e same keys can be in a different order than the input array. To solve this problem, we can iterate the input array in reverse order to build the output array. This strategy helps us to keep the same keys in the same order as they appear in the input array.

A screenshot of a number

Description automatically generated

1. Sort based on the tens place digits

A screenshot of a cell phone

Description automatically generated

1. Sort based on hundreds place digits

Time Complexity: O (d x (n + b)), where:

* d – Number of digits in the largest number
* b – Base (10 for decimal numbers)
* n – Number of elements

Overall complexity: O(n)

Space Complexity: O(n)

void countSort(int arr[], int n, int exp)

{

    // Output array

    int output[n];

    int i, count[10] = { 0 };

    // Store count of occurrences

    // in count[]

    for (i = 0; i < n; i++)

        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i]

    // now contains actual position

    // of this digit in output[]

    for (i = 1; i < 10; i++)

        count[i] += count[i - 1];

    // Build the output array

    for (i = n - 1; i >= 0; i--) {

        output[count[(arr[i] / exp) % 10] - 1] = arr[i];

        count[(arr[i] / exp) % 10]--;

    }

    // Copy the output array to arr[],

    // so that arr[] now contains sorted

    // numbers according to current digit

    for (i = 0; i < n; i++)

        arr[i] = output[i];

}

// The main function to that sorts arr[]

// of size n using Radix Sort

void radixsort(int arr[], int n)

{

    // Find the maximum number to

    // know number of digits

    int m = getMax(arr, n);

    // Do counting sort for every digit.

    // Note that instead of passing digit

    // number, exp is passed. exp is 10^i

    // where i is current digit number

    for (int exp = 1; m / exp > 0; exp \*= 10)

        countSort(arr, n, exp);

}