Data Structure and Algorithms Notes

Contents

[Linked Lists: 2](#_Toc181110483)

[Constructor: 3](#_Toc181110484)

[Destructor: 3](#_Toc181110485)

[Traversal: 4](#_Toc181110486)

[Searching: 4](#_Toc181110487)

[Finding Length: 5](#_Toc181110488)

[Insertion: 6](#_Toc181110489)

[Insertion at Beginning: 6](#_Toc181110490)

[Insertion at End: 6](#_Toc181110491)

[Insertion at Specific Position: 8](#_Toc181110492)

[Deletion: 9](#_Toc181110493)

[Deletion at Beginning: 9](#_Toc181110494)

[Deletion at the End 10](#_Toc181110495)

[Deletion of Specific Position: 12](#_Toc181110496)

[Stacks: 14](#_Toc181110497)

[Constructor: 16](#_Toc181110498)

[Destructor: 16](#_Toc181110499)

[Push() – Insert element into stack: 17](#_Toc181110500)

[Pop() – Remove element from top of stack: 18](#_Toc181110501)

[Peek() – Returns top element from stack: 20](#_Toc181110502)

[isEmpty() – Returns true if empty, else false 21](#_Toc181110503)

[append() – Adds to the end of the stack 21](#_Toc181110504)

[Queues: 22](#_Toc181110505)

[Constructor: 24](#_Toc181110506)

[Destructor: 24](#_Toc181110507)

[Enqueue: 25](#_Toc181110508)

[Dequeue: 25](#_Toc181110509)

[getFront() 27](#_Toc181110510)

[getRear() 27](#_Toc181110511)

[isEmpty() 27](#_Toc181110512)

# 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:**

* **Simplicity:**Stacks are a simple and easy-to-understand data structure, making them suitable for a wide range of applications.
* **Efficiency:**Push and pop operations on a stack can be performed in constant time **(O(1))**, providing efficient access to data.
* **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.
* **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()**to insert an element into the stack
3. **pop()**to remove an element from the stack
4. **peek()**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

    // Function to push a new element onto the stack

    void push(int data) {

        StackNode\* stackNode = new StackNode(data); // Create a new node

        stackNode->next = root;                     // Link new node to the top of the stack

        root = stackNode;                           // Update root to the new node

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

    }

## 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.

    int pop() {

        if (isEmpty()) {                            // Check if stack is empty

            cout << "Stack is empty\n";

            return INT\_MIN;                         // Return a minimum integer if stack is empty

        }

        StackNode\* temp = root;                     // Store the current top node in temp

        root = root->next;                          // Move root to the next node in the stack

        int popped = temp->data;                    // Retrieve the data of the popped node

        delete temp;                                // Delete the popped node

        return popped;                              // Return the popped element's data

    }

## Peek() – 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**.

    int peek() const {

        if (isEmpty()) {                            // Check if stack is empty

            cout << "Stack is empty\n";

            return INT\_MIN;                         // Return a minimum integer if stack is empty

        }

        return root->data;                          // Return data of the top element

    }

## 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 Data Structure is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end (back of the queue), and all deletions from the list are made at the other end (front of the queue). The element, which is first pushed into the order, the delete operation is first performed on that.

A diagram of a queue data structure

Description automatically generated

**Advantages of Queue Data Structure:**

* A large amount of data can be managed efficiently with ease.
* Operations such as insertion and deletion can be performed with ease as it follows the first in first out rule.
* Queues are useful when a particular service is used by multiple consumers.
* Queues are fast in speed for data inter-process communication.
* Queues can be used in the implementation of other data structures.

**Disadvantages of Queue Data Structure:**

* The operations such as insertion and deletion of elements from the middle are time consuming.
* Searching an element takes O(N) time.
* Maximum size of a queue must be defined prior in case of array implementation.

**Basic Operations in Queue Data Structure:**

1. Constructor
2. Destructor
3. enqueue(): Adds (or stores) an element to the end of the queue.
4. dequeue(): Removal of elements from the queue.
5. peek() or getFront(): Acquires the data element available at the front node of the queue without deleting it.
6. getRear(): This operation returns the element at the rear end without removing it.
7. isEmpty: Checks if the queue is empty.

## Constructor:

Constructor just sets front and rear pointers to be nullptr

        Time Complexity: O(1)

        Space Complexity: O(1)

  Queue() {

        front = rear = nullptr;

    }

## Destructor:

Cleans up all dynamically allocated nodes to prevent memory leaks when Queue object is destroyed

Calls dequeue in a while loop

Time Complexity: O(n), where n is the number of nodes in the queue.

Space Complexity: O(1)

    ~Queue() {

        while (!isEmpty()) {

            dequeue();  // Continuously dequeue until the queue is empty, ensuring all nodes are deleted

        }

    }

## Enqueue:

Adds a node to the back of the queue

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

        Space Complexity: O(1) - Allocates space for one new node regardless of queue size.

A diagram of a number

Description automatically generated A diagram of a number

Description automatically generated

* Create a new node
* If queue is empty, new node becomes front and rear
* Otherwise, add new node at end of queue and update rear

  void enqueue(int new\_data) {

        // Create a new linked list node with the provided data

        Node\* new\_node = new Node(new\_data);

        // If queue is empty, the new node becomes both front and rear

        if (this->isEmpty()) {

            front = rear = new\_node;

            return;

        }

        // Otherwise, add the new node at the end of the queue and update rear

        rear->next = new\_node;

        rear = new\_node;

    }

## Dequeue:

Removes a node at the start of the queue

        Time Complexity: O(1) - Constant time as it only removes the front node.

        Space Complexity: O(1) - Uses a constant amount of extra space for temporary storage.

A screenshot of a diagram

Description automatically generatedA screenshot of a diagram

Description automatically generated

* If queue is empty, output underflow message error
* Store current front node as temp pointer
* Move front to the next node in queue
* If front becomes nullptr after moving, set rear to null ptr as well (this is for when queue is empty)
* Delete the temp node

This code, you might want to adjust to return the value the node stored, instead of returning void

    void dequeue() {

        // If queue is empty, output underflow message and return

        if (this->isEmpty()) {

            cout << "Queue Underflow\n";

            return;

        }

        // Store the current front node in a temporary pointer

        Node\* temp = front;

        // Move front to the next node in the queue

        front = front->next;

        // If front becomes nullptr after moving, set rear to nullptr as well

        if (front == nullptr)

            rear = nullptr;

        // Deallocate memory of the old front node

        delete temp;

    }

## getFront()

Peek to get the value (or data) from the front of the queue

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

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

    int getFront() const {

        // Check if the queue is empty

        if (this->isEmpty()) {

            cout << "Queue is empty\n";

            return INT\_MIN;

        }

        return front->data;    // Return the data of the front node

    }

## getRear()

Retrieve the value from the rear of the queue

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

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

   int getRear() const {

        // Check if the queue is empty

        if (this->isEmpty()) {

            cout << "Queue is empty\n";

            return INT\_MIN;

        }

        return rear->data;     // Return the data of the rear node

    }

## isEmpty()

Boolean check to see if queue is empty

        Time Complexity: O(1) - Constant time as it only checks the front pointer.

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

    bool isEmpty() const

    {

        // If the front is null, then the queue is empty; otherwise, it's not

        return front == nullptr;

    }