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:**

* **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(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

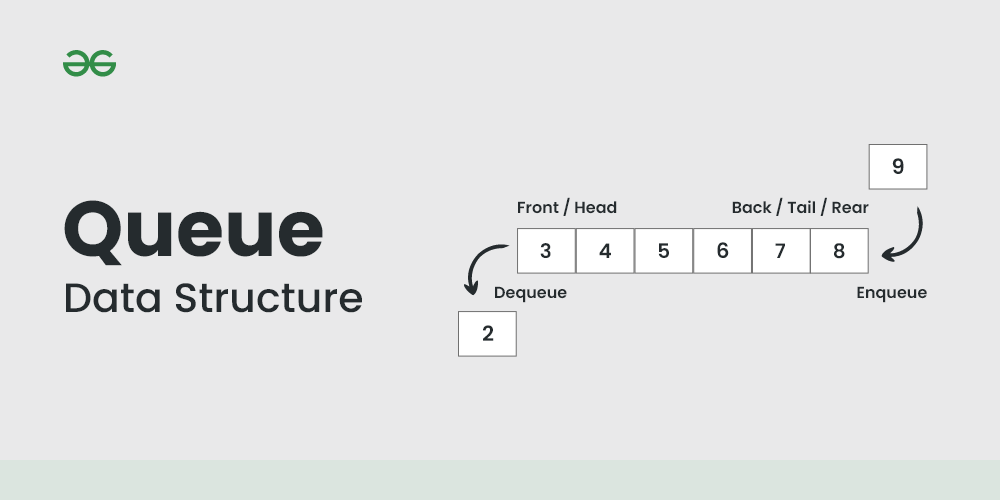
        }

        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;

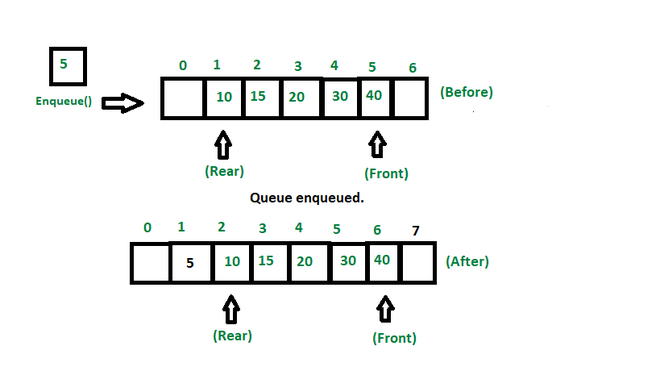
}

## 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;

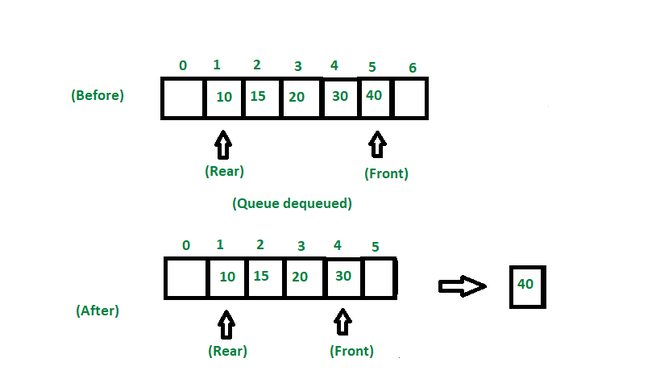
}

## 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;

        }

    }

}