Exam_Review_2

Recursion

Requirements: be able to write simple recursive functions

Definition: defining a problem in terms of itself

> *A function that calls upon itself*

Example:

```C++
// power function
unsigned int pow(int x, int n){
        if(n==0){return 1;}
        else if(n<0){
                return 1/(x * pow(x, n-1));
        }
        else {
                return (x * pow(x, n-1));
        }
}

// factorial funciton
int fact(int x){
        if(n==1){return 1;}
        else return x*(fact(x-1));
}
```

Math Review

Induction: We need to prove n = 1, as well as the LHS implies the RHS when n = n+1. Prove P(n)⇒P(n+1).

Example: For all natural numbers n, the sum of the first n natural numbers is equal to (n · (n + 1)) / 2.

Basis Step: n = 1
LHS = 1

RHS = (1 · (1 + 1)) / 2 = 2 / 2 = 1

Inductive Hypothesis: Assume that the statement is true for some arbitrary positive integer n.
1 + 2 + 3 + ... + n = (n · (n + 1)) / 2

Inductive Step: Prove that the statement is also true for k + 1.

1. $1 + 2 + 3 + \ldots + k + (k + 1)$
2. $(k \cdot (k + 1) + 2(k + 1))/2$
3. = $(k^2 + k + 2k + 2)/2$
4. = $(k^2 + 3k + 2)/2$
5. = $((k + 1) \cdot (k + 2))/2$

C++ Basic Concepts

Member's

Name	Definition
Public	Accessible from anywhere, assuming the object is within scope
Private	Accessible only from within the class scope
Protected	The same protections as private, however the this data is available within derived classes as well

Parameter Passing Methods

Name	Desc	Modify	Issue	Benefit
Pass by value	A copy is passed to function	Does not affect original argument	Overhead of Copy	
Pass by reference	Passes reference to argument	Directly affects original		Avoids copy
Pass by Pointer	Passes pointer to actual argument	Directly affects original		Doesn't copy and allows for null ptr
Pass by Const Ref	Passes const ref to argument	Cannot modify Value		Avoid Copy
Pass by Const Value	Similar to pass by value	Cannot modify argument		Ensures unchanged
Pass by Rvalue	Transfer of ownership or modification of temp object	Efficient operations		

Examples

```C++
void passByValue(int x) {
    x = 10; // Changes to x won't affect the original value
}

void passByReference(int& x) {
    x = 10; // Changes to x affect the original value
}

void passByPointer(int* ptr) {
    *ptr = 10; // Changes to the value pointed to by ptr affect the original value
}

void passByConstReference(const int& x) {
    // Cannot modify x, but avoids copying for large objects
}

void passByConstValue(const int x) {
    // Cannot modify x
}

void moveSemantics(int&& x) {
    // x is an rvalue reference
}
```

Return Passing

Name	Desc	Use
Return by Value	returns a copy of value or object	Returning small simple datatypes/object
Return by Reference	returns a reference to value or object	Modify value outside of function
Return by Pointer	returns a pointer to value or object	Return pointer/indicate potential failure
Return by Const Reference	returns const reference to value or object	Returning values that should/cannot be modified
Return by Rvalue Reference	returns a r value reference to value or object	Transferring ownership, or efficiently returning temporary objects
Return by struct	Allows you to return multiple values	touple / struct

Examples

```cpp
int returnByValue() {
    return 42; // Returns a copy of the integer 42
}

int& returnByReference(int& x) {
    return x; // Returns a reference to the integer x
}

int* returnByPointer(int* ptr) {
    return ptr; // Returns a pointer to the integer pointed to by
ptr
}

const int& returnByConstReference(int& x) {
    return x; // Returns a constant reference to the integer x
}

int&& returnByRvalueReference() {
    return std::move(someTempObject); // Returns an rvalue reference
}

std::tuple<int, double> returnByTuple() {
    return std::make_tuple(42, 3.14); // Returns a tuple with two
values
}
```

Generic Programming

Template Functions: represents a type parameter that is filled in with a specific data type when you use the function.

```cpp
template <typename T>
T maximum(T a, T b) {
        return (a > b) ? a : b;
}
```

The above function creates an template T and allows for passing of:

1. Objects

2. Primitive Types (int, char, etc)
3. Pointers
4. References
5. Enums
6. Arrays
7. Function Pointers
8. Member Function Pointers
9. ETC

Algorithm Analysis

Definition: Algorithm analysis is the process of evaluating the efficiency and performance of algorithms. It helps determine how an algorithm's execution time and resource usage (e.g., memory) grow as the size of the input data increases. It provides insights into the algorithm's scalability and helps make informed choices about which algorithm to use for a particular problem.

Type	Description	Represents
Big O	Stating that the algorithms running time is at most proportional to a function of the input size	Upper bounds
Big Omega	Stating that the algorithms running time is at least proportional to a function of the input size	Lower bounds
Big Theta	Stating that the algorithms running time grows at the same rate as a function of the input size	Average (Both Upper and Lower)

Order of Important Functions

Order	Time Complexity	Name
1	O(1)	Constant time
2	O(log n)	Logarithmic time
3	O(n)	Linear time
4	O(n log n)	Linearithmic time
5	O(n^2)	Quadratic time
6	O(2^n)	Exponential time
7	O(n!)	Factorial time

Determining Complexity of Functions/Algorithms

1. Count the basic operation in the algorithm

2. Express the count as a function of the input size (use n)
3. Simplify the function if plausible
4. Analyze worst case (Big O), average case (Big Theta), and the best case (Big Omega).

A Few Data Structures

Prototypes: refer to the high-level, abstract descriptions or blueprints of data structures.

Structure Name	Operations	Properties	Use Cases	Complexity
Stack	push, pop, top, is_empty	LIFO (Last-In-First-Out)	Function call tracking, expression evaluation	Typically O(1)
Vector	push_back, pop_back, size, capacity	Resizable, order collection	Dynamic lists, array like behavior	Push and Pop back O(1)
Tree	insert, delete, search, traversal	Hierarchical, organized structure	Representing hierarchical data, searching, sorting	Depends on the specific type of tree (e.g., O(log n) for balanced binary search trees).
Array	Indexing, insertion, deletion	Ordered collection of elements	Sequential access data storage	Indexing is O(1), but insertion/deletion can be O(n) in the worst case.
Queue	enqueue, dequeue, is_empty	FIFO (First-In-First-Out) order	Task scheduling, breadth-first search	O(1) for enqueue and dequeue.
Linked List	insert, delete, search, traversal	Dynamic, node-based structure	Efficient insertion/deletion, memory management	Depends on the specific type, but can be O(1) for some operations.
Hash Set	insert, erase, find	Unordered collection with fast access	Fast lookup, eliminating duplicates	O(1) average case for most operations, but can be slower in case of collisions.

Doubly Linked List Implementation

```cpp
#include <iostream>

template <typename T>
class DoublyLinkedList {
private:
    struct Node {
        T data;
        Node* next;
        Node* prev;
        Node(const T& value) : data(value), next(nullptr),
prev(nullptr) {}
    };

    Node* head;
    Node* tail;

public:
    DoublyLinkedList() : head(nullptr), tail(nullptr) {}

    // Insert an element at the end of the list
    void insert(const T& value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
    }

    // Erase an element with the specified value
    void erase(const T& value) {
        Node* current = head;
        while (current) {
            if (current->data == value) {
                if (current->prev) {
                    current->prev->next = current->next;
                } else {
```

```
                head = current->next;
            }
            if (current->next) {
                current->next->prev = current->prev;
            } else {
                tail = current->prev;
            }
            delete current;
            return;
        }
        current = current->next;
    }
}

// Iterator class for the doubly-linked list
class Iterator {
private:
    Node* current;

public:
    Iterator(Node* node) : current(node) {}

    T& operator*() {
        return current->data;
    }

    Iterator& operator++() {
        current = current->next;
        return *this;
    }

    Iterator& operator--() {
        current = current->prev;
        return *this;
    }

    bool operator!=(const Iterator& other) {
        return current != other.current;
    }
};
```

```
    Iterator begin() {
        return Iterator(head);
    }

    Iterator end() {
        return Iterator(nullptr);
    }
};
```

Double-ended Queue

```C++
#include <iostream>
#include <stdexcept>

template <typename T>
class Deque {
private:
    T* elements;
    int capacity;
    int size;
    int front;
    int back;

public:
    Deque(int initial_capacity = 10) : capacity(initial_capacity),
size(0), front(0), back(0) {
        elements = new T[capacity];
    }

    ~Deque() {
        delete[] elements;
    }

    bool empty() const {
        return size == 0;
    }

    int count() const {
        return size;
    }

    void push_front(const T& value) {
```

```
        if (size == capacity) {
            resize();
        }
        front = (front - 1 + capacity) % capacity;
        elements[front] = value;
        size++;
    }

    void push_back(const T& value) {
        if (size == capacity) {
            resize();
        }
        elements[back] = value;
        back = (back + 1) % capacity;
        size++;
    }

    void pop_front() {
        if (empty()) {
            throw std::out_of_range("Deque is empty.");
        }
        front = (front + 1) % capacity;
        size--;
    }

    void pop_back() {
        if (empty()) {
            throw std::out_of_range("Deque is empty.");
        }
        back = (back - 1 + capacity) % capacity;
        size--;
    }

    T& operator[](int index) {
        if (index < 0 || index >= size) {
            throw std::out_of_range("Index out of bounds.");
        }
        return elements[(front + index) % capacity];
    }

    T& front_element() {
```

```
        if (empty()) {
            throw std::out_of_range("Deque is empty.");
        }
        return elements[front];
    }

    T& back_element() {
        if (empty()) {
            throw std::out_of_range("Deque is empty.");
        }
        return elements[(back - 1 + capacity) % capacity];
    }

    class Iterator {
    private:
        Deque& deque;
        int index;

    public:
        Iterator(Deque& d, int i) : deque(d), index(i) {}

        T& operator*() {
            return deque[index];
        }

        Iterator& operator++() {
            index = (index + 1) % deque.size;
            return *this;
        }

        bool operator!=(const Iterator& other) {
            return index != other.index;
        }
    };

    Iterator begin() {
        return Iterator(*this, 0);
    }

    Iterator end() {
        return Iterator(*this, size);
```

```
    }

private:
    void resize() {
        int new_capacity = capacity * 2;
        T* new_elements = new T[new_capacity];
        for (int i = 0; i < size; i++) {
            new_elements[i] = elements[(front + i) % capacity];
        }
        front = 0;
        back = size;
        capacity = new_capacity;
        delete[] elements;
        elements = new_elements;
    }
};
```

Complexity of key methods:

- Accessing elements by index (`[]`): O(1)
- Accessing front or back elements (`front_element` and `back_element`): O(1)
- Pushing and popping elements from the front or back (`push_front`, `pop_front`, `push_back`, and `pop_back`): O(1) amortized (