Lecture 02: Elementary Data Structures (Arrays, Linked Lists, Stacks, Queues)

C++ Code Samples — Sedgwick Algorithms Course — lecture-02-samples.cpp

```cpp
// ============================================================================
// Lecture 02: Elementary Data Structures
// Sedgwick Algorithms Course
//
// Topics covered:
//   - Dynamic array (resizing array) implementation
//   - Singly linked list with insert, delete, print
//   - Stack using a linked list (push, pop, peek)
//   - Queue using a linked list (enqueue, dequeue)
//   - Balanced parentheses checker using a stack
// ============================================================================

#include <iostream>
#include <string>
#include <stdexcept>

using namespace std;

// === SECTION: Dynamic Array (Resizing Array) ===
// Doubles capacity when full, halves when quarter full.
// This is the strategy used by java.util.ArrayList and std::vector.
class DynamicArray {
private:
    int* data;
    int size;
    int cap;

    void resize(int newCap) {
        cout << "    [resize " << cap << " -> " << newCap << "]\n";
        int* newData = new int[newCap];
        for (int i = 0; i < size; ++i) newData[i] = data[i];
        delete[] data;
        data = newData;
        cap = newCap;
    }

public:
    DynamicArray() : size(0), cap(1), data(new int[1]) {}
    ~DynamicArray() { delete[] data; }

    void push_back(int val) {
        if (size == cap) resize(2 * cap);  // double when full
        data[size++] = val;
    }

    int pop_back() {
        if (size == 0) throw underflow_error("empty");
        int val = data[--size];
        // shrink when quarter full to maintain amortized O(1)
        if (size > 0 && size == cap / 4) resize(cap / 2);
        return val;
    }

    int peek() const { return data[size - 1]; }
    int get(int i) const { return data[i]; }

    void print() const {
        cout << "    [";
        for (int i = 0; i < size; ++i) {
            if (i > 0) cout << ", ";
            cout << data[i];
        }
```

```
        cout << "] (size=" <<  << ", capacity=" <<  << ")\n"
    }
};

// === SECTION: Singly Linked List ===
// Each node stores a value and a pointer to the next node.
// Insertions/deletions at the head are O(1).
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int v, ListNode* n = nullptr) : val(v), next(n) {}
};

class SinglyLinkedList {
private:
    ListNode* head;
    int sz;

public:
    SinglyLinkedList() : head(nullptr), sz(0) {}

    ~SinglyLinkedList() {
        while (head) { ListNode* t = head; head = head->next; delete t; }
    }

    // Insert at the front: O(1)
    void insertFront(int val) {
        head = new ListNode(val, head);
        ++sz;
    }

    // Insert at a specific position (0-indexed): O(n)
    void insertAt(int pos, int val) {
        if (pos == 0) { insertFront(val); return; }
        ListNode* cur = head;
        for (int i = 0; i < pos - 1 && cur; ++i) cur = cur->next;
        if (!cur) return;
        cur->next = new ListNode(val, cur->next);
        ++sz;
    }

    // Delete first occurrence of val: O(n)
    bool deleteVal(int val) {
        if (!head) return false;
        if (head->val == val) {
            ListNode* t = head; head = head->next; delete t; --sz;
            return true;
        }
        ListNode* cur = head;
        while (cur->next && cur->next->val != val) cur = cur->next;
        if (!cur->next) return false;
        ListNode* t = cur->next;
        cur->next = cur->next->next;
        delete t;
        --sz;
        return true;
    }

    int size() const { return sz; }

    void print(const string& label = "") const {
        if (!label.empty()) cout << label << ": ";
        else cout << "    ";
```

```
        ListNode* cur = head;
        while (cur) {
            cout << cur->val;
            if (cur->next) cout << " -> ";
            cur = cur->next;
        }
        cout << " (size=" << sz << ")\n";
    }
};

// === SECTION: Stack (Linked List Implementation) ===
// LIFO: Last In, First Out. All operations are O(1).
class Stack {
private:
    ListNode* top_;
    int sz;

public:
    Stack() : top_(nullptr), sz(0) {}
    ~Stack() { while (top_) { ListNode* t = top_; top_ = top_->next; delete t; } }

    void push(int val) { top_ = new ListNode(val, top_); ++sz; }

    int pop() {
        if (!top_) throw runtime_error("stack underflow");
        int val = top_->val;
        ListNode* t = top_;
        top_ = top_->next;
        delete t;
        --sz;
        return val;
    }

    int peek() const {
        if (!top_) throw runtime_error("stack empty");
        return top_->val;
    }

    bool empty() const { return top_ == nullptr; }
    int size() const { return sz; }
};

// === SECTION: Queue (Linked List Implementation) ===
// FIFO: First In, First Out. Enqueue at tail, dequeue from head. Both O(1).
class Queue {
private:
    ListNode* head;
    ListNode* tail;
    int sz;

public:
    Queue() : head(nullptr), tail(nullptr), sz(0) {}
    ~Queue() { while (head) { ListNode* t = head; head = head->next; delete t; } }

    void enqueue(int val) {
        ListNode* node = new ListNode(val);
        if (!tail) { head = tail = node; }
        else { tail->next = node; tail = node; }
        ++sz;
    }

    int dequeue() {
        if (!head) throw runtime_error("queue empty");
        int val = head->val;
```

```
        ListNode* t = head;
        head = head->next;
        if (!head) tail = nullptr;
        delete t;
        --sz;
        return val;
    }

    int front() const {
        if (!head) throw runtime_error("queue empty");
        return head->val;
    }

    bool empty() const { return head == nullptr; }
    int size() const { return sz; }
};

// === SECTION: Balanced Parentheses Checker ===
// Classic stack application. Push opening brackets, pop and match on closing.
bool isBalanced(const string& s) {
    Stack st;
    for (char c : s) {
        if (c == '(' || c == '[' || c == '{') {
            st.push(c);
        } else if (c == ')' || c == ']' || c == '}') {
            if (st.empty()) return false;
            char top = st.pop();
            if ((c == ')' && top != '(') ||
                (c == ']' && top != '[') ||
                (c == '}' && top != '{'))
                return false;
        }
    }
    return st.empty();
}

// === MAIN ===
int main() {
    cout << "==========================================\n";
    cout << " Lecture 02: Elementary Data Structures\n";
    cout << "==========================================\n";

    // --- Dynamic Array ---
    cout << "\n--- Dynamic Array (Resizing Array) ---\n";
    DynamicArray da;
    for (int i = 1; i <= 10; ++i) {
        da.pushBack(i * 10);
    }
    da.printAll();
    cout << "  Popping 5 elements...\n";
    for (int i = 0; i < 5; ++i) {
        cout << "    popped: " << da.popBack() << "\n";
    }
    da.printAll();

    // --- Singly Linked List ---
    cout << "\n--- Singly Linked List ---\n";
    SinglyLinkedList list;
    list.insertFront(30);
    list.insertFront(20);
    list.insertFront(10);
    list.printAll("After inserting 10, 20, 30 at front");
    list.insertAt(1, 15);   // insert 15 at position 1
    list.printAll("After inserting 15 at position 1");
```

```
    list.deleteVal(20);
    list.print("After deleting 20");

    // --- Stack ---
    cout << "\n--- Stack (Linked List) ---\n";
    Stack stk;
    cout << "  Pushing: 10, 20, 30\n";
    stk.push(10); stk.push(20); stk.push(30);
    cout << "  Peek: " << stk.peek() << "\n";
    cout << "  Pop:  " << stk.pop() << "\n";
    cout << "  Pop:  " << stk.pop() << "\n";
    cout << "  Pop:  " << stk.pop() << "\n";
    cout << "  Stack empty: " << (stk.empty() ? "yes" : "no") << "\n";

    // --- Queue ---
    cout << "\n--- Queue (Linked List) ---\n";
    Queue q;
    cout << "  Enqueue: 10, 20, 30\n";
    q.enqueue(10); q.enqueue(20); q.enqueue(30);
    cout << "  Front:   " << q.front() << "\n";
    cout << "  Dequeue: " << q.dequeue() << "\n";
    cout << "  Dequeue: " << q.dequeue() << "\n";
    cout << "  Dequeue: " << q.dequeue() << "\n";
    cout << "  Queue empty: " << (q.empty() ? "yes" : "no") << "\n";

    // --- Balanced Parentheses ---
    cout << "\n--- Balanced Parentheses Checker (Stack Application) ---\n";
    string tests[] =
    {
        "((()))"          // balanced
        "{[()]}"          // balanced
        "({[])"           // NOT balanced (mismatched)
        "((("             // NOT balanced (unclosed)
        "a*(b+c)-{d/[e]}" // balanced (ignores non-bracket chars)
        ""                // balanced (empty string)
    };
    for (const auto& t : tests)
    {
        cout << "  \"" << t << "\" -> "
             << (isBalanced(t) ? "BALANCED" : "NOT BALANCED") << "\n";
    }

    return 0;
}
```