Homework #1

CE 6320: Applied Data Structures and Algorithms (Fall 2025)

University of Texas at Dallas

Instructor: Prof. Mohamed Ibrahim
Due Date: September 16, 2025

Student Name: _____

Student ID: _____

Instructions:

- This homework has **10 problems** and **1 bonus problem**.
- You are required to **solve and submit any 5 problems of your choice** out of the 10.
- Each chosen problem will be graded out of its assigned points; your total will be scaled to an overall maximum of **100 points**.
- The **bonus problem** is optional. Points earned there will be added on top of the 100 points as extra credit.
- Show all steps clearly. Partial credit will be awarded.

September 9, 2025

Problem 1: Circular Linked List — Playlist Rotation **[20 points]**

Problem Description. A playlist of n songs labeled $1..n$ is stored as a *singly circular linked list*. The last node points back to the first node, forming a circle. A pointer `cur` always marks the "current song". Two basic operations are defined:

- `NEXT(k)`: move `cur` forward by k steps (wrapping around if needed).
- `DEL()`: delete the node pointed to by `cur`, and set `cur` to the next node.

—

1. Diagram (5 pts). Draw a circular linked list for $n = 5$. Explain clearly how the nodes are allocated in memory (heap vs stack), how they are connected by next-pointers, and how `cur` points to the "current" node. Illustrate what happens when the node with value 3 is deleted.

—

2. C++ Implementation (9 pts). Implement `Node` and fill in the bodies of the following functions using "raw pointers" in C++:

```
// Defines the node of the linked list
struct Node {
    // ...
};

// Builds a circular list of 1..n nodes, returns pointer to head,
// initializes cur appropriately, and sets size = n.
Node* buildCircle(int n, Node*& cur, int& size);

// Advances cur by k % size steps.
void nextK(Node*& cur, int& size, int k);

// Deletes the node at cur, updates links, and sets cur = next node.
void delCurrent(Node*& cur, int& size);

// Prints the list once around starting from cur (or EMPTY if size = 0).
void printOnce(Node* cur);
```

—

3. Tracking (4 pts). Start with $n = 5$ and `cur` $= 1$. Apply the following sequence of operations:

`NEXT(2)`, `DEL()`, `NEXT(3)`, `DEL()`, `NEXT(1)`.

After each operation, draw the circular linked list and indicate which node `cur` is pointing to. At the end, show the sequence printed by `printOnce`.

—

4. Complexity (2 pts). Compute the time complexity of each function you implemented (`buildCircle`, `nextK`, `delCurrent`, `printOnce`) as a function of k and n.

Problem 2: Unrolled Linked List **[20 points]**

Problem Description. An *unrolled linked list* is a linked list where each node (called a *block*) stores multiple elements inside a small fixed-size array. This makes the list shorter (fewer pointers) and improves cache performance compared to a regular linked list where each node stores just one element.

Each block can hold up to B elements. For example, if $B = 4$ and we want to store the sequence $[1, 2, 3, 4, 5, 6, 7, 8]$, the structure looks like this:

$$[1, 2, 3, 4] \quad \rightarrow \quad [5, 6, 7, 8]$$

If we insert one more element (say, 9) into the second block, it overflows. In that case, the block is *split* into two blocks, roughly half-and-half:

$$[1, 2, 3, 4] \quad \rightarrow \quad [5, 6] \quad \rightarrow \quad [7, 8, 9]$$

Similarly, if we delete elements and a block becomes too empty (fewer than half full), it may either *borrow* an element from a neighboring block or *merge* with its neighbor.

We will use 0-based indexing across the entire list: the first element of the first block has index 0, the second has index 1, and so on.

—

1. Diagram (5 pts). Let $B = 4$. Draw an unrolled linked list that stores the sequence $[1, 2, 3, 4, 5, 6, 7, 8]$. Show the blocks, their arrays (with used slots grouped at the front), each block's `cnt` (number of items), and the next-pointers. Explain what lives on the heap (blocks and arrays) versus what lives on the stack (local variables, pointers).

—

2. C++ Implementation (9 pts). Using raw pointers and a fixed capacity $B = 4$, fill in the function bodies below. Implement insertions with block-split when full, and deletions with borrow/merge when too empty.

```
const int B = 4;

struct Block {
    int a[B];      // items packed in a[0..cnt-1]
    int cnt;       // number of valid items in this block
    Block* next;   // pointer to next block
};

struct UList {
    Block* head;   // head block (nullptr if empty)
    int n;         // total number of items
    UList(): head(nullptr), n(0) {}
};

// Build list from arr[0..n-1], filling blocks left-to-right.
void buildFromArray(const int* arr, int n, UList& L);

// Return the value at global index idx.
```

```
int getAt(const UList& L, int idx);

// Insert x at global index idx, shifting items as needed.
// If a block is full, split it.
void insertAt(UList& L, int idx, int x);

// Erase the element at global index idx, shifting items as needed.
// If a block becomes under half full, borrow or merge with neighbor.
void eraseAt(UList& L, int idx);

// Helper: find block containing index idx, return pointer and local offset.
Block* findBlock(UList& L, int idx, int& off);
```

—

3. Tracking (4 pts). Let $B = 4$. Start with $[1, 2, 3, 4, 5, 6, 7, 8]$. Apply the operations below (indices are global, 0-based):

`insertAt(2, 99)`, `insertAt(6, 77)`, `eraseAt(3)`, `insertAt(8, 55)`, `eraseAt(0)`.

After each operation, redraw the blocks showing their contents, counts, and next-pointers. Indicate where a **split**, **borrow**, or **merge** takes place.

—

4. Complexity (2 pts). Let the total number of items be n and block capacity be B. Give Big-O bounds for:

`buildFromArray`, `getAt`, `insertAt`, `eraseAt`.

Express your answers in terms of n and B, and briefly justify each.

Problem 3: LRU Cache **[20 points]**

Problem Description. An LRU (Least Recently Used) cache stores up to a fixed number (C) of key–value pairs. When the cache is full and a new key must be inserted, the *least recently used* key is evicted. On every successful access (read or write), that key becomes *most recently used*.

In this problem, keys are integers in the range $[0, K)$. We implement LRU without hashing by using: (1) a **doubly linked list** that maintains recency order (head = most recent, tail = least recent), and (2) an **array directory** `dir`$[0..K-1]$ that maps each key to its node (or `nullptr` if absent).

The cache supports:

- `get(key)` → returns the value if present (and moves key to the front), or "not found,"
- `put(key, val)` → inserts/updates; if full and inserting a new key, evict the LRU (tail).

—

1. Diagram (5 pts). Draw a memory diagram for a cache with $K = 10$ and capacity $C = 3$ after the operations `put(2,20)`, `put(5,50)`, `put(7,70)`. Show:

- the doubly linked list order (head = MRU on the left, tail = LRU on the right),
- which objects live on the heap (nodes, the directory array) vs. on the stack (local variables),
- the directory array `dir` entries that are non-`nullptr` and what they point to.

—

2. C++ Implementation (9 pts). Fill in the bodies of the following functions using raw pointers (no hashing). Keys are in $[0, K)$.

```
struct Node {
    int key, val;
    Node *prev, *next;
    Node(int k, int v): key(k), val(v), prev(nullptr), next(nullptr) {}
};

struct LRU {
    int cap;        // capacity C
    int size;       // current number of keys in cache
    int K;          // key range upper bound (keys in [0, K))
    Node **dir;     // array directory: dir[key] = Node* or nullptr
    Node *head;     // most recently used (MRU)
    Node *tail;     // least recently used (LRU)
    LRU(): cap(0), size(0), K(0), dir(nullptr), head(nullptr), tail(nullptr) {}
};

// Initialize the cache with capacity Ccap and key range K.
// Allocate dir on the heap and set all entries to nullptr.
void initLRU(LRU &C, int Ccap, int K);

// Return true and set outVal if present (and move node to MRU); else return false.
bool get(LRU &C, int key, int &outVal);

// Insert or update (key, val). If inserting and size==cap, evict LRU.
// Move the (possibly new) key to MRU position.
```

```
void put(LRU &C, int key, int val);

// (Helpers students should implement/and may call from get/put)
// Remove node u from list in O(1).
void detach(LRU &C, Node *u);

// Insert node u at head (MRU) in O(1).
void pushFront(LRU &C, Node *u);

// Evict the LRU (tail) node in O(1); update dir and size; delete the node.
void evictLRU(LRU &C);
```

—

3. Tracking (4 pts). Let $K = 10$ and capacity $C = 3$. Starting from an empty cache, apply:

`put(2,20)`, `put(5,50)`, `put(7,70)`, `get(2)`, `put(9,90)`, `get(5)`, `put(2,21)`.

After each operation, show:

- the list from MRU (head) to LRU (tail) as $[k{:}v, \ldots]$,
- which key (if any) was evicted,
- any changes in the directory entries you deem relevant.

—

4. Complexity (2 pts). Give tight Big-O time bounds (as functions of K and C) for `initLRU`, `get`, `put`. State the auxiliary space usage (in terms of K and C) and justify briefly.

Problem 4: Polynomial Representation & Recursive Evaluation [20 points]

Problem Description. We represent a polynomial

$$P(x) = \sum_i c_i x^{e_i}$$

as a *singly linked list* of terms, each term storing a coefficient c_i and an integer exponent e_i. Terms are kept in **strictly descending** exponent order (largest exponent first). Example (descending exponents): $3x^5 + 2x^3 - x + 4$ is stored as the list

$$(3, 5) \to (2, 3) \to (-1, 1) \to (4, 0).$$

When evaluating, note that exponents can be *sparse* (e.g., there is no x^2 term above). A convenient recursive strategy is to "bridge" the exponent gap between consecutive terms using a power of x. For instance,

$$3x^5 + 2x^3 = \left(3x^{5-3} + 2\right) \cdot x^3 = (3x^2 + 2) \cdot x^3.$$

—

1. Diagram (5 pts). Using the example $P(x) = 3x^5 + 2x^3 - x + 4$, draw the linked list at the level of nodes and pointers. Label each node with $(\text{coef}, \text{exp})$. Explain what lives on the heap (nodes) and what lives on the stack (local pointers, temporaries) during typical operations.

—

2. C++ Implementation (9 pts). Fill in the bodies of the following functions using raw pointers. Maintain the list in **strictly descending** exponent order with **no duplicate exponents**. When inserting a term whose exponent already exists, **combine** coefficients; if the new coefficient becomes 0, remove that node.

```
struct Term {
    double coef;
    int exp;
    Term* next;
    Term(double c, int e): coef(c), exp(e), next(nullptr) {}
};

// Build a polynomial from arrays coef[i], exp[i], i=0..m-1.
// Insert terms one-by-one so that the final list is strictly descending in exp
// and contains no duplicate exponents (combine coefs; drop zero terms).
Term* buildFromArrays(const double* coef, const int* exp, int m);

// Insert (c,e) into the sorted list (descending exponents).
// Combine coefficients if exp already exists; remove node if coef becomes 0.
void insertTerm(Term*& head, double c, int e);

// Remove the term with exponent e, if present.
void eraseExp(Term*& head, int e);

// Recursively compute x^k for k>=0 using fast exponentiation.
// Must run in O(log k).
double powInt(double x, int k);
```

```
// Recursive evaluation of the polynomial at x.
// Assume list is strictly descending in exp with no duplicates.
// Hint: if current node is (c,e) and next is (c2,e2), multiply the running
// value by x^(e - e2) before adding c2, and recurse.
double evalRecursive(const Term* head, double x);
```

Notes. (i) Do not use `std::vector`, `std::map`, or sorting libraries; work with raw pointers. (ii) You may write small private helpers if needed (e.g., to combine or remove nodes). (iii) Handle corner cases: empty list, single node, insertion at head/tail, deletion at head.

—

3. Tracking (4 pts). Start from arrays

$$\texttt{coef} = [3,\ 2,\ -1,\ 4], \quad \texttt{exp} = [5,\ 3,\ 1,\ 0]$$

and build $P(x)$. Then apply, in order:

$$\texttt{insertTerm}(+5,\ 3), \quad \texttt{eraseExp}(1), \quad \texttt{insertTerm}(-3,\ 5).$$

After each operation, redraw the linked list (node contents and arrows), indicating any nodes that were combined or removed. Finally, evaluate the resulting polynomial at $x = 2$ using your `evalRecursive`. Show the key recursive steps: the exponent gaps you raise x to, and the partial values as they propagate back up.

—

4. Complexity (2 pts). Let n be the number of *nodes* (nonzero terms) in the list, and let Δ_i be the exponent gap between consecutive terms during evaluation. Give tight Big-O bounds for:

`buildFromArrays`, `insertTerm`, `eraseExp`, `evalRecursive`.

Express `evalRecursive` in terms of n and the cost of powers (i.e., $\sum_i \log \Delta_i$ using `powInt`).

Problem 5: Filesystem Flattening **[20 points]**

Problem Description. We model a tiny filesystem as a *multilevel linked list*. Each node represents either a directory or a file. Siblings within the same directory are connected by `next`, and a directory's first entry (its sublist of children) is reached via `child`. Your goal is to *flatten* the hierarchy into a single-level list using `next` pointers only, in **preorder** (visit a directory, then its entire subtree, left-to-right), clearing all `child` pointers in the result.

Example. Arrows = `next`, downward arrow = `child`:

$$1\ (\text{dir}) \ \downarrow\ \big(2\ (\text{file}) \ \rightarrow\ 3\ (\text{dir}) \ \rightarrow\ 4\ (\text{file})\big), \quad 3 \downarrow \big(5\ (\text{file}) \ \rightarrow\ 6\ (\text{file})\big)$$

Preorder flattening produces: $1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 4$, with all `child` pointers set to `nullptr`.

—

1. Diagram (5 pts). Draw the illustrative structure above. Clearly label `next` vs `child` pointers in your diagram. Explain what lives on the heap (nodes) and what lives on the stack (local variables, recursion frames).

—

2. C++ Implementation (9 pts). Fill in the bodies of the following functions using raw pointers. **Requirement: `flatten` must be implemented *recursively***, in-place (reuse nodes; do not allocate new nodes just to rearrange). It must produce preorder and set all `child` pointers in reachable nodes to `nullptr` in the result.

```
struct Node {
    int id;       // unique ID (e.g., 1..n)
    bool isDir;   // true = directory, false = file
    Node* next;   // next sibling (or next in flattened list)
    Node* child;  // first child if directory, else nullptr
    Node(int i, bool d): id(i), isDir(d), next(nullptr), child(nullptr) {}
};

// Append 'kid' at the end of directory 'dir' child-list.
void addChild(Node* dir, Node* kid);

// Append 'sib' after node 'u' in the sibling chain (attach at the end of u's next-chain).
void appendSibling(Node* u, Node* sib);

// Recursively flatten the multilevel list rooted at 'head' into a single-level
// preorder list. After flatten, all child pointers in reachable nodes must be nullptr.
// Return the head of the flattened list.
Node* flatten(Node* head);

// Print IDs once along 'next' starting at head (e.g., "10 11 12 ...")
// or "EMPTY" if head==nullptr.
void printFlat(Node* head);
```

—

3. Tracking (4 pts). Build the following multilevel list (IDs shown; arrows = `next`, downward = `child`):

Node 10 (dir)
children: 11 (file) → 12 (dir) → 13 (file)
12 has children: 14 (dir) → 16 (file)
14 has child: 15 (file)

Now call `flatten` on Node 10. After flattening, draw the resulting single-level list (only `next` pointers) and give the exact printout produced by `printFlat`. Then, argue briefly that all `child` pointers reachable from Node 10 are `nullptr` after flattening.

—

4. Complexity (2 pts). Let n be the number of nodes reachable from the head and d be the maximum directory nesting depth. Give tight Big-O bounds for `addChild`, `appendSibling`, `flatten`, and `printFlat` as functions of n. State the recursion depth in terms of d.

Problem 6: Weather Forecasting **[20 points]**

Problem Description. You are building a tiny forecasting module that processes a stream of hourly temperatures (integers). For each hour i, with window size W (e.g., last 24 hours), you must compute:

- the rolling **mean** over the last W values,
- the rolling **minimum** and **maximum** over the last W values,
- a naive **next-hour forecast** F_{i+1} defined as the current rolling mean.

To be efficient in streaming, you will maintain: (1) a *window queue* of the last W values using an array-backed *ring buffer*; (2) a *monotonic increasing deque* for the window minimum; (3) a *monotonic decreasing deque* for the window maximum; (4) a running *sum* for the mean.

Example. Suppose $W = 3$ and the temperatures arrive as $[7, 5, 9, 6]$. After reading 7: window $= \{7\}$, mean $= 7$, min $= 7$, max $= 7$. After reading 5: window $= \{7, 5\}$, mean $= 6$, min $= 5$, max $= 7$. After reading 9: window $= \{7, 5, 9\}$, mean $= 7$, min $= 5$, max $= 9$, forecast $F_3 = 7$. After reading 6: window slides to $\{5, 9, 6\}$, mean $= 20/3$, min $= 5$, max $= 9$, forecast $F_4 = 20/3$.

—

1. Diagram (5 pts). For $W = 4$ and processed values $[10, 3, 8, 12]$, draw:

- the ring buffer showing `head`, `tail`, and contents (in array indices),
- the *increasing* deque state (store values, front is the current minimum),
- the *decreasing* deque state (store values, front is the current maximum).

Explain what lives on the heap (the arrays for ring buffer and deques) vs on the stack (local counters, pointers).

—

2. C++ Implementation (9 pts). Fill in the bodies using raw pointers and fixed-capacity arrays allocated on the heap. Maintain $O(1)$ amortized time per incoming value.

```
struct RingBuffer {
    int *a; int cap, head, tail, cnt;
    RingBuffer(int W);           // allocate a[W], initialize fields
    ~RingBuffer();               // delete[]
    void push(int x);            // push to tail (assume cnt < cap)
    int  pop();                  // pop from head (assume cnt > 0); return value
    bool full()  const;
    bool empty() const;
};

struct MonoDequeInc {            // increasing deque for window min
    int *a; int cap, L, R;       // store values; indices [L, R)
    MonoDequeInc(int W);
    ~MonoDequeInc();
    void push(int x);            // pop-back while back > x, then push x
    void popIf(int x);           // if front == x, pop-front
    int  getMin() const;         // front
};
```

```
struct MonoDequeDec {            // decreasing deque for window max
    int *a; int cap, L, R;       // store values; indices [L, R)
    MonoDequeDec(int W);
    ~MonoDequeDec();
    void push(int x);            // pop-back while back < x, then push x
    void popIf(int x);           // if front == x, pop-front
    int  getMax() const;         // front
};

struct ForecastWindow {
    int W; RingBuffer q; MonoDequeInc qmin; MonoDequeDec qmax;
    long long sum;
    ForecastWindow(int W);
    void ingest(int x, bool &windowReady); // push x; if q.full() before ingest, evict oldest
    double mean() const;                   // sum / W (assume windowReady)
    int    wmin() const;                   // qmin.getMin()
    int    wmax() const;                   // qmax.getMax()
    double forecastNext() const;           // equal to mean()
};
```

Notes. (i) Implement ring indices with modular arithmetic; (ii) In `ingest`, if the window already holds W items, first evict the oldest: update `sum`, `qmin.popIf(old)`, `qmax.popIf(old)`, then push the new value: update `sum`, `qmin.push(x)`, `qmax.push(x)`, `q.push(x)`. (iii) For the deques, store *values* (not indices); `popIf` removes the front iff it equals the evicted value.

—

3. Tracking (4 pts). Let $W = 3$. Process the stream $[6, 4, 7, 3, 5]$ in order. After each ingest:

- show the ring buffer (`head`, `tail`, contents),
- show the increasing deque (state from front to back) and the decreasing deque (front to back),
- if the window is full, report mean, min, max, and the forecast (equals the mean).

—

4. Complexity (2 pts). Let n be the number of ingested values and W the window size. Give tight Big-O bounds for total processing time and auxiliary space of your implementation.

Problem 7: Postfix Evaluation in Circuit Simulation [20 points]

Problem Description. In a simple circuit simulator, equivalent component values (e.g., resistances) are often composed using *series* and *parallel* operators. To streamline evaluation, expressions are written in *postfix* (Reverse Polish) notation and evaluated with a stack.

We will use the following tokens:

- **Operands**:
  - Numeric constants, e.g., `"10"`, `"3.5"`.
  - Variables `"R0"`..`"R9"` referring to an array `R[10]` of component values (e.g., resistances).
- **Operators**:
  - `'S'`: series, Series$(a, b) = a + b$.
  - `'P'`: parallel, Parallel$(a, b) = (a^{-1} + b^{-1})^{-1}$.
  - `'+'`, `'-'`, `'*'`, `'/'`: standard arithmetic.
  - `' '`: unary negation.

Clarifying Example. Suppose we want the equivalent resistance of two resistors $R_0 = 10$ and $R_1 = 5$ in parallel, then add a 2-ohm resistor in series. In postfix this is written:

[`"R0"`, `"R1"`, `"P"`, `"2"`, `"S"`].

Stack trace:

- Push `R0`=10 ⇒ stack [10].
- Push `R1`=5 ⇒ stack [10, 5].
- Apply `P`: pop 5 and 10, compute $10 \parallel 5 = (1/10 + 1/5)^{-1} = 3.33$, push result ⇒ stack [3.33].
- Push 2 ⇒ stack [3.33, 2].
- Apply `S`: pop 2 and 3.33, compute $3.33 + 2 = 5.33$, push result ⇒ stack [5.33].

Final result = 5.33 ohms. This is the kind of evaluation your program must automate for any valid postfix expression.

—

1. Diagram (5 pts). Draw the array-backed *stack* you will use for evaluation (top index, capacity, contents). Explain clearly what is on the heap (the stack's underlying array) and what is on the stack (local variables, indices). Give a small illustration: evaluate the example above and show the stack contents after each token.

—

2. C++ Implementation (9 pts). Using raw pointers and no STL containers beyond C-strings, fill in the function bodies below. Assume tokens are provided as C-strings `const char* tokens[]` and the variable table `R[10]` is given.

```
struct DStack {
    double *a; int top; int cap;
    DStack(int C);    // allocate a[C] on the heap; set top = 0
    ~DStack();        // delete[] a
    bool empty() const;
    bool full()  const;
    void push(double x);   // assume not full
```

```
    double pop();          // assume not empty
    double peek() const;   // assume not empty
};

// Parse helpers (return true if parsed successfully).
bool parseConst(const char* tok, double& out);   // decimal constant
bool parseVarR (const char* tok, const double R[10], double& out); // "Rk"

bool isUnaryOp (const char* tok, char& op);      // "~"
bool isBinaryOp(const char* tok, char& op);      // '+','-','*','/','S','P'

double applyUnary (char op, double x);           // "~": -x
double applyBinary(char op, double a, double b); // order: a (left), b (right)

double evalPostfix(const char* tokens[], int m, const double R[10]);
```

Requirements.

- Evaluate strictly left-to-right over tokens, using the stack.
- For a binary operator, *pop right operand first* (b), then left (a), then push `applyBinary`(op, a, b).
- For parallel `'P'`, assume inputs > 0.
- You may ignore malformed inputs; assume expressions are valid and stack never overflows for the given tests.

—

3. Tracking (4 pts). Let `R` = [10, 5, 2, 4, ...] so that `R0` = 10, `R1` = 5, `R2` = 2, `R3` = 4. Evaluate the postfix sequence:

["R0", "R1", "P", "R2", "S", "R3", "S"].

After each token, show the entire numeric stack (from bottom to top). Give the final numeric result.

—

4. Complexity (2 pts). Let m be the number of tokens. State tight Big-O bounds for time and auxiliary space of your evaluator as functions of m. Explain briefly.

Problem 8: Two Stacks in One Array **[20 points]**

Problem Description. You will implement two independent stacks that *share a single array.* Stack 1 grows from the *left* (index 0 upward), and Stack 2 grows from the *right* (last index downward). This maximizes space utilization: as long as there is at least one free cell between the two tops, pushes can proceed. An overflow happens only when the two tops are about to collide (no free cell remains).

Example (indices 0..5). Let the array length be $N = 6$. Initially `top1` $= -1$ and `top2` $= 6$.

- `push1(10)` places 10 at index 0, so `top1` becomes 0.
- `push2(70)` places 70 at index 5, so `top2` becomes 5.
- `push1(20)` places 20 at index 1; `push2(60)` places 60 at index 4; `push1(30)` places 30 at index 2.
- `pop2()` removes from index 4; then `push2(50)` places 50 at index 4; `push2(40)` places 40 at index 3.

This example illustrates how the two stacks grow toward each other while sharing space.

—

1. Diagram (5 pts). Draw the single array for $N = 6$ with indices 0..5 beneath the cells. Show the positions of `top1` and `top2` *after each operation* in the sequence:

`push1(10)`, `push2(70)`, `push1(20)`, `push2(60)`, `push1(30)`, `pop2()`, `push2(50)`, `push2(40)`.

Clearly indicate which cells belong to Stack 1 vs Stack 2. Explain briefly how overflow is detected in this design.

—

2. C++ Implementation (9 pts). Using raw pointers and *one* heap-allocated array, fill in the bodies of the functions below. Do not use STL containers. Assume integers for stack items.

```
struct TwoStacks {
    int *a;   // single array on the heap of length n
    int n;    // capacity
    int top1; // last used index of Stack 1  (starts at -1)
    int top2; // first free index past Stack 2 (starts at n)
    TwoStacks(int N);   // allocate a[N], set n=N, top1=-1, top2=N
    ~TwoStacks();       // delete[] a

    bool full()  const; // no free cell remains: (top1 + 1 == top2)
    bool empty1() const;
    bool empty2() const;

    void push1(int x);  // pre: not full
    void push2(int x);  // pre: not full
    int  pop1();        // pre: not empty1
    int  pop2();        // pre: not empty2
    int  peek1() const; // pre: not empty1
    int  peek2() const; // pre: not empty2
};
```

Requirements.

- Stack 1 grows left→right: `++top1; a[top1]=x;`.
- Stack 2 grows right→left: `-top2; a[top2]=x;`.
- `full()` is true iff `top1 + 1 == top2`.
- `pop1()` returns `a[top1-]`; `pop2()` returns `a[top2++]`.
- Assume test cases avoid invalid operations (overflow/underflow) unless explicitly asked.

—

3. Tracking (4 pts). Let $N = 8$. Starting from an empty structure, apply:

`push2(90)`, `push1(11)`, `push1(22)`, `push2(80)`, `push1(33)`, `push2(70)`, `pop1()`, `push2(60)`, `push1(44)`.

After each operation, draw the array, label `top1` and `top2`, and show the contents. State whether the *next* operation `push1(55)` would overflow, and justify your answer.

—

4. Complexity (2 pts). For each of the following functions, state the tight time complexity and auxiliary space usage as functions of N:

`push1`, `push2`, `pop1`, `pop2`, `peek1`, `peek2`, `full`, `empty1`, `empty2`.

Provide brief justifications.

Problem 9: Real-Time Task Scheduling with a Ring Buffer Deque [20 points]

Problem Description. In a small real-time executive, runnable tasks are dispatched in short time slices. To accommodate both urgent and background work, the scheduler uses a *double-ended queue (deque)* so it can insert and remove tasks at either end:

- urgent tasks are enqueued at the *front* (high priority, sooner dispatch),
- background tasks are enqueued at the *back* (lower priority),
- a preempted task may be returned to the *front*,
- the least-important queued task may be dropped from the *back* under memory pressure.

The deque is implemented as a **ring buffer** (circular array) for $O(1)$ amortized operations and predictable memory use.

Example. Let the queue hold `Task{id}` items and start empty with capacity $C = 5$. Operations (left = front, right = back):

$$\texttt{ENQ\_HIGH(10)} \Rightarrow [\,10\,], \quad \texttt{ENQ\_LOW(20)} \Rightarrow [\,10 \mid 20\,], \quad \texttt{ENQ\_HIGH(11)} \Rightarrow [\,11, 10 \mid 20\,].$$

A dispatch step removes from the *front*: `DISPATCH()` yields 11 and the queue becomes $[\,10 \mid 20\,]$. If `PREEMPT(10)` occurs, put 10 back to the *front*: $[\,10 \mid 20\,] \rightarrow [\,10, 10 \mid 20\,]$ (illustrative; in the assignment below, `PREEMPT(id)` enqueues a *given* id at the front).

1. Diagram (5 pts). Draw a ring-buffer deque of capacity $C = 6$ (array indices 0..5) after each operation in the sequence below. Show the positions of `head`, `tail`, and the current `count`. Indicate which indices are occupied and which are free.

`ENQ_LOW(21)`, `ENQ_LOW(22)`, `ENQ_HIGH(11)`, `ENQ_LOW(23)`, `DISPATCH()`, `ENQ_HIGH(12)`, `DROP_LOW()`.

Explain briefly what lives on the heap (the deque array) vs. on the stack (local variables like indices, temporary task objects).

2. C++ Implementation (9 pts). Using raw pointers and a single heap-allocated array, fill in the bodies of the following. Your deque must store `Task` objects and support all operations in $O(1)$ amortized time.

```
struct Task {
    int id;            // unique task identifier
    // you may add small fields if needed (e.g., stamp), but not STL containers
};

struct DequeRB {
    Task* a;           // circular array on the heap
    int cap;           // capacity (fixed at construction)
    int head;          // index of front element (valid iff count > 0)
    int tail;          // index one past the back element (in ring space)
    int count;         // number of elements currently stored

    DequeRB(int C);    // allocate a[C], set fields for empty deque
    ~DequeRB();        // delete[] a

    bool empty() const;
    bool full()  const;
```

```
    // push to front/back; pre: not full
    void pushFront(const Task& t);
    void pushBack (const Task& t);

    // pop from front/back; pre: not empty
    Task popFront();
    Task popBack ();

    // accessors; pre: not empty
    Task& front();
    Task& back ();
};

// Simple scheduler facade using the deque.
// ENQ_HIGH -> pushFront, ENQ_LOW -> pushBack,
// DISPATCH -> popFront (returns dispatched Task id)
// PREEMPT(id) -> pushFront(Task{id}),
// DROP_LOW() -> popBack().
struct Scheduler {
    DequeRB q;

    Scheduler(int C);

    void ENQ_HIGH(int id);
    void ENQ_LOW (int id);
    int  DISPATCH();       // returns id of dispatched Task
    void PREEMPT(int id);  // place given id back at front
    void DROP_LOW();       // remove one Task from the back
};
```

Notes.

- Implement ring arithmetic with modular indices; do not use STL containers.
- When `full()`, you may assume inputs avoid further enqueues unless asked (no need to resize).
- All operations should avoid shifting elements (use index arithmetic to wrap around).

3. Tracking (4 pts). Let capacity $C = 5$. Starting from an empty scheduler, apply:

`ENQ_LOW(30)`, `ENQ_HIGH(12)`, `ENQ_LOW(31)`, `ENQ_HIGH(13)`, `DISPATCH()`, `PREEMPT(99)`, `ENQ_LOW(32)`, `DROP_L`

After *each* operation:

- draw the ring buffer showing `head`, `tail`, `count`, and occupied indices,
- write the deque order from *front to back* as a list of task IDs,
- for each `DISPATCH()`, report the returned task ID.

4. Complexity (2 pts). For your `DequeRB` and `Scheduler`, state tight Big-O bounds (in terms of capacity C and number of operations m) for:

`pushFront`, `pushBack`, `popFront`, `popBack`, `front`, `back`, `ENQ_HIGH`, `ENQ_LOW`, `DISPATCH`, `PREEMPT`, `DROP_LOW`.

Explain briefly and state the auxiliary space used by your implementation.

Problem 10: Particle Simulation: SoA vs. AoS **[20 points]**

Problem Description. You are simulating N point particles in 2D with positions (x_i, y_i) and velocities $(v_{x,i}, v_{y,i})$. At each time step Δt, you advance

$$x_i \leftarrow x_i + v_{x,i}\,\Delta t, \qquad y_i \leftarrow y_i + v_{y,i}\,\Delta t.$$

Two memory layouts are common:

- **AoS (Array of Structures):** one array of `Particle` where each element stores `{x,y,vx,vy}` together.
- **SoA (Structure of Arrays):** four parallel arrays `x[]`, `y[]`, `vx[]`, `vy[]` storing each field separately.

Example. Let $N = 3$. AoS stores:

$$\texttt{P[0]} = \{x_0, y_0, vx_0, vy_0\},\ \texttt{P[1]} = \{x_1, y_1, vx_1, vy_1\},\ \texttt{P[2]} = \{x_2, y_2, vx_2, vy_2\}.$$

SoA stores:

$$\texttt{x} = [x_0, x_1, x_2], \quad \texttt{y} = [y_0, y_1, y_2], \quad \texttt{vx} = [vx_0, vx_1, vx_2], \quad \texttt{vy} = [vy_0, vy_1, vy_2].$$

A single update step adds $\Delta t \cdot$ `vx[i]` to `x[i]` and $\Delta t \cdot$ `vy[i]` to `y[i]` for each i. This assignment compares the two layouts in code, tracing, and complexity.

—

1. Diagram (5 pts). Draw both layouts for $N = 4$:

(a) **AoS:** show the contiguous array of `Particle` elements and the field order inside each element.

(b) **SoA:** show the four separate arrays and how index i refers to one particle across arrays.

For each layout, explain what is allocated on the **heap** (arrays and/or array of structs) and what lives on the **stack** (local pointers, loop indices, temporaries).

—

2. C++ Implementation (9 pts). Using raw pointers (no STL containers), fill in the bodies of the following functions for *both* AoS and SoA. Assume doubles for all components, and that memory is heap-allocated in the init/build functions.

```
// ----- AoS -----
struct Particle {
    double x, y, vx, vy;
};

struct AoS {
    Particle* p;   // array length N on the heap
    int N;
};

// Allocate AoS of size N and initialize from arrays (x[], y[], vx[], vy[]).
void aosInit(AoS& A, int N, const double* x, const double* y,
             const double* vx, const double* vy);

// Advance positions by dt: x += vx*dt, y += vy*dt for all particles.
```

```
void aosStep(AoS& A, double dt);

// Compute center of mass: returns (cx, cy) via refs.
void aosCenterOfMass(const AoS& A, double& cx, double& cy);

// Free AoS heap memory.
void aosFree(AoS& A);

// ----- SoA -----
struct SoA {
    double *x, *y, *vx, *vy;  // four arrays length N on the heap
    int N;
};

// Allocate SoA of size N and initialize from arrays (x[], y[], vx[], vy[]).
void soaInit(SoA& S, int N, const double* x, const double* y,
             const double* vx, const double* vy);

// Advance positions by dt: x[i] += vx[i]*dt, y[i] += vy[i]*dt.
void soaStep(SoA& S, double dt);

// Compute center of mass: returns (cx, cy) via refs.
void soaCenterOfMass(const SoA& S, double& cx, double& cy);

// Free SoA heap memory.
void soaFree(SoA& S);

// ----- Layout transform (optional helper pair) -----
// Copy A -> S (same N); Copy S -> A (same N). Assume targets allocated.
void aosToSoa(const AoS& A, SoA& S);
void soaToAos(const SoA& S, AoS& A);
```

Notes. (i) Assume inputs are valid and memory allocation succeeds. (ii) Use simple for-loops; do not use SIMD or libraries. (iii) Avoid temporary allocations inside tight loops.

—

3. Tracking (4 pts).

Let $N = 3$ with initial arrays

$$\texttt{x} = [1, -2, 0], \quad \texttt{y} = [0, 3, -1], \quad \texttt{vx} = [2, -1, 4], \quad \texttt{vy} = [-3, 0, 1], \qquad \Delta t = 0.5.$$

(a) Show the AoS memory picture *before* the step and *after* one call to `aosStep(A, dt)` (values for all fields).

(b) Show the SoA memory picture *before* the step and *after* one call to `soaStep(S, dt)` (values in each array).

(c) Using your functions, compute and report the center of mass (c_x, c_y) after the step for each layout.

—

4. Complexity (2 pts). Let N be the number of particles. State the time complexity and auxiliary space of each function (`aosInit`, `aosStep`, `aosCenterOfMass`, `aosFree`, `soaInit`,

`soaStep`, `soaCenterOfMass`, `soaFree`, and optionally `aosToSoa`/`soaToAos`) as functions of N. Briefly explain any differences in memory access behavior between AoS and SoA for the `Step` and `CenterOfMass` operations.

Problem 11: Bonus — Real-Time Timer Wheel [25 points]

Problem Description. You are building a millisecond-resolution timer service for a tiny real-time runtime. Timers can be *one-shot* (fire once after a delay) or *periodic* (fire every fixed period). To avoid per-timer scans, the runtime uses a *timer wheel*: a circular array (*ring*) of W *slots*, where each slot holds a singly linked list of timers scheduled to (potentially) fire when the wheel's *cursor* lands on that slot. For delays $\geq W$, each timer stores a nonnegative integer `rounds` counting how many full revolutions remain before it can fire. Time advances in discrete ticks; each tick moves the cursor by $+1 \bmod W$.

Example. Let $W = 8$, start time $t = 0$, cursor at slot 0.

- Schedule one-shot T_1 with delay 3: place in slot $(0 + 3) \bmod 8 = 3$ with `rounds` $= 0$.
- Schedule one-shot T_2 with delay 13: slot $(0 + 13) \bmod 8 = 5$, `rounds` $= \lfloor 13/8 \rfloor = 1$.
- Schedule periodic T_3 with period 4 (next fire in 4): slot 4, `rounds` $= 0$, and remember period 4.

Tick to $t = 1$: cursor 1 (no fire). $t = 2$: cursor 2 (no fire). $t = 3$: cursor 3 $\Rightarrow$ process slot 3: T_1 fires (and is removed). $t = 4$: cursor 4 $\Rightarrow$ T_3 fires; since it is periodic, it is rescheduled with delay 4 into slot 0 (cursor is 4, so $(4+4) \bmod 8 = 0$), `rounds` $= 0$. At $t = 5$: cursor 5 $\Rightarrow$ process slot 5: T_2 has `rounds` $= 1$, so decrement to 0 and keep it in slot 5 for the next revolution (no fire yet).

—

1. Diagram (5 pts). Let $W = 8$. Draw the timer wheel as a ring labeled slots 0..7 and indicate the *cursor*. For each slot, show a vertical singly linked list of timers (by ID) enqueued there, and for each timer show its stored `rounds` and (if applicable) `period`. Briefly explain what lives on the **heap** (the slots array, timer nodes, optional directory) vs. on the **stack** (local pointers, counters).

—

2. C++ Implementation (9 pts). Using raw pointers (no STL containers), fill in the bodies below. Each slot stores a forward list of `Timer` nodes. Cancelling uses an optional *directory* array for $O(1)$ lookup by bounded IDs $[0, K)$; if you choose not to use a directory, your `cancel` may scan the wheel.

```
struct Timer {
    int id;          // unique in [0, K)
    int rounds;      // revolutions remaining before eligible to fire
    int period;      // period (>0) if periodic; 0 if one-shot
    Timer* next;     // next timer in the slot's singly linked list
    Timer(int i=0,int r=0,int p=0): id(i), rounds(r), period(p), next(nullptr) {}
};

struct Bucket { Timer* head; Bucket(): head(nullptr) {} };

struct TimerWheel {
    Bucket* slots;   // array of W buckets on the heap
    int W;           // number of slots
    int cursor;      // current slot index [0..W-1]
    long long t;     // ticks since start
    // Optional directory for O(1) cancel by id; size K, entries are Timer* or nullptr.
    Timer** dir;     // may be nullptr to indicate "unused"
```

```
    int K;           // directory size (0 if dir==nullptr)

    TimerWheel(): slots(nullptr), W(0), cursor(0), t(0), dir(nullptr), K(0) {}
};

// Allocate W buckets (and directory of size K if K>0), initialize fields.
void initWheel(TimerWheel& TW, int W, int K);

// Schedule a timer relative to "now": if period>0 -> periodic, else one-shot.
// delay >= 0 measured in ticks.
void addTimer(TimerWheel& TW, int id, int delay, int period);

// Cancel timer by id; return true if found and removed, false otherwise.
bool cancelTimer(TimerWheel& TW, int id);

// Advance time by exactly one tick: move cursor, process the bucket.
// For each timer in the current slot:
//   - if rounds>0: decrement and keep it
//   - else (rounds==0): fire; if periodic, reschedule with 'period'; if one-shot, remove.
// Collect fired ids into out[] up to maxOut; set firedCount to number stored.
void tick1(TimerWheel& TW, int* out, int& firedCount, int maxOut);

// Advance by 'steps' ticks by repeatedly calling tick1 (convenience).
void tickN(TimerWheel& TW, int steps, int* out, int& firedCount, int maxOut);

// (Optional small helpers you may implement and call)
// Compute (slot, rounds) from a nonnegative delay relative to current cursor.
void mapDelay(const TimerWheel& TW, int delay, int& slot, int& rounds);

// Reinsert 'tm' at front of bucket 'b' (or in any order you choose).
void pushFront(Bucket& b, Timer* tm);

// Remove a timer with given id from a single bucket list (returns Timer* or nullptr).
Timer* removeFromBucket(Bucket& b, int id);
```

Requirements.

- Use modular arithmetic for cursor movement and slot mapping.
- `tick1` must run in time proportional to the number of timers in the current bucket (plus any reschedules), not total timers.
- If you implement the directory, keep it consistent on add/reschedule/cancel; set `dir[id]=nullptr` when the timer is removed.
- Assume inputs avoid overflow of the output buffer `out[]` (you may truncate if needed).

—

3. Tracking (4 pts). Let $W = 6$, $K \geq 1000$, start at $t = 0$, cursor $= 0$. Apply the following in order (IDs in parentheses):

`addTimer`(100, 2, 0), `addTimer`(200, 7, 0), `addTimer`(300, 3, 3),
`tick1`, `tick1`, `tick1`, `tick1`, `cancelTimer`(300), `tickN`(3),
`addTimer`(400, 1, 0), `tickN`(2).

After *each* operation:

- draw the wheel: mark cursor position and list the timers (with their current `rounds`) in each slot,
- list fired timer IDs in the order they fired during that step (if any),
- state the contents of the directory entry `dir[id]` for IDs $100, 200, 300$ (or write "(unused)" if no directory is implemented).

—

4. Complexity (2 pts). Let n be the number of live timers, W the number of slots, and let k_s be the number of timers stored in slot s when processed. State tight Big-O bounds (and justify briefly) for:

`initWheel`, `addTimer`, `cancelTimer` (with and without directory), `tick1` (in terms of k_{cursor}), `tickN`.

Also state the auxiliary space in terms of n, W, and K (if using a directory).