Fonction pour ajouter un élément en fin de liste

```
cel* ajouter_fin(cel* T, int valeur) {
    cel* nouv = malloc(sizeof(cel));
    nouv->val = valeur;
    nouv->next = NULL;

    if (T == NULL) {
        return nouv;
    }

    cel* p = T;
    while (p->next != NULL) {
        p = p->next;
    }
    p->next = nouv;
    return T;
}
```

Fonction pour afficher liste

```
void afficher(cel* T) {
    cel* p = T;
    while (p != NULL) {
        printf("%d -> ", p->val);
        p = p->next;
    }
    printf("NULL\n");
}
```

main()

```
cel* T = NULL;

    int valeur, continuer;

    printf("Ajout des elements a la liste:\n");
    do {

        printf("Entrer une valeur : ");
        scanf("%d", &valeur);

        T = ajouter_fin(T, valeur);

        printf("Voulez-vous ajouter une autre valeur ? (1: Oui / 0: Non): ");
        scanf("%d", &continuer);


    } while (continuer == 1);
```

Function to find the maximum value recursively

```
int max_recursive(cel *T) {
    if (T == NULL) {
        return -1; // Assuming all values in the list are positive integers
    }
    if (T->next == NULL) {
        return T->val;
    }

    int max_suivant = max_recursive(T->next);
    return (T->val > max_suivant) ? T->val : max_suivant;
}
```

Function to find the maximum value iteratively

```
int max_iterative(cel *T) {

    if (T == NULL) {
        printf("La liste est vide.\n");
        return -1;
    }

    int max_val = T->val;
    cel *p = T->next;
    while (p != NULL) {
        if (p->val > max_val) {
            max_val = p->val;
        }
        p = p->next;
    }

    return max_val;
}
```

Fonction pour rattacher L2 à la suite de L1

```
cel* rattacher_listes(cel* L1, cel* L2) {

    if (L1 == NULL) {
        return L2; // Si L1 est vide, retourner L2
    }

    cel* p = L1;
    while (p->next != NULL) {
        p = p->next; // Trouver le dernier élément de L1
    }

    p->next = L2; // Rattacher L2 à la fin de L1
    return L1;
```

Fonction pour séparer une liste en deux listes : positifs et négatifs

```
void separer_listes(cel* T, cel** positifs, cel** negatifs) {

    cel* p = T;

    while (p != NULL) {
        if (p->val >= 0) {
            *positifs = ajouter_fin(*positifs, p->val);
        } else {
            *negatifs = ajouter_fin(*negatifs, p->val);
        }
        p = p->next;
    }
}

int main() {
    cel* T = NULL;
    cel* positifs = NULL;
    cel* negatifs = NULL;
    int valeur, continuer;

    separer_listes(T, &positifs, &negatifs);
```

déchanger les positions de deux cellules données par les pointeurs `t` et `v`

```
void echanger_positions(cel** T, cel* t, cel* v) {

    if (t == v || t == NULL || v == NULL) return; // Cas inutiles

    cel* prevT = NULL, *prevV = NULL, *p = *T;

    // Trouver les précédents de t et v
    while (p != NULL && (prevT == NULL || prevV == NULL)) {
        if (p->next == t) prevT = p;
        if (p->next == v) prevV = p;
        p = p->next;
    }

    // Si t ou v est la tête, ajuster directement
    if (*T == t) prevT = NULL;
    if (*T == v) prevV = NULL;

    // Ajuster les chaînages
    if (prevT) prevT->next = v;
    if (prevV) prevV->next = t;

    cel* temp = t->next;
    t->next = v->next;
    v->next = temp;

    // Ajuster la tête si nécessaire
    if (*T == t) *T = v;
    else if (*T == v) *T = t;

}
```

- **`supprimer_occurrences`** pour supprimer toutes les occurrences d'un élément donner

- **`garder_k_occurrences`** pour conserver seulement les `k` premières occurrences d'un élément.
- **`supprimer_duplicats`** pour ne garder que la première occurrence de chaque élément.

```
cel* supprimer_occurrences(cel* T, int x) {
    cel *p = T, *prev = NULL;
    while (p != NULL) {
        if (p->val == x) {
            if (prev == NULL) {
                T = p->next;
                free(p);
                p = T;
            } else {
                prev->next = p->next;
                free(p);
                p = prev->next;
            }
        } else {
            prev = p;
            p = p->next;
        }
    }
    return T;
}

cel* garder_k_occurrences(cel* T, int x, int k) {
    cel *p = T, *prev = NULL;
    int count = 0;

    while (p != NULL) {
        if (p->val == x) {
            count++;
            if (count > k) {
                if (prev == NULL) {
                    T = p->next;
                    free(p);
                    p = T;
                } else {
                    prev->next = p->next;
                    free(p);
                    p = prev->next;
                }
                continue;
            }
        }
        prev = p;
        p = p->next;
    }
    return T;
}
```

```c
cel* supprimer_duplicats(cel* T) {
    cel *p = T;
    while (p != NULL) {
        T = garder_k_occurrences(T, p->val, 1);
        p = p->next;
    }
    return T;
}
```

Fonction pour inverser la liste de maniere iterative

```c
cel* inverser_iteratif(cel* T) {
    cel *precedent = NULL, *courant = T, *suivant = NULL;

    while (courant != NULL) {
        suivant = courant->next;   // Sauvegarder le suivant
        courant->next = precedent; // Inverser le pointeur
        precedent = courant;       // Avancer precedent
        courant = suivant;         // Avancer courant
    }

    return precedent; // Nouveau tete de la liste
}
```

Fonction pour inverser la liste de maniere recursive

```c
cel* inverser_recursif(cel* T) {
    if (T == NULL || T->next == NULL) {
        return T;      // Cas de base : liste vide ou un seul element
    }

    cel* reste = inverser_recursif(T->next);  // Inverser le reste de la liste
    T->next->next = T;    // Faire pointer le suivant vers l'actuel
    T->next = NULL;        // Terminer la liste a l'actuel

    return reste;    // Nouveau tete de la liste
}
```

Fonction pour transformer une liste lineaire en liste circulaire

```
cel* transformer_circulaire(cel* T) {
    if (T == NULL) {
        printf("La liste est vide. Rien a transformer.\n");
        return NULL;
    }

    cel* p = T;
    while (p->next != NULL) {
        p = p->next;
    }
    p->next = T; // Le dernier noeud pointe vers le premier
    return T;
}
```

Fonction pour afficher une liste chainee circulaire

```
void afficher_circulaire(cel* T) {
    if (T == NULL) {
        printf("La liste est vide.\n");
        return;
    }

    cel* p = T;
    do {
        printf("%d -> ", p->val);
        p = p->next;
    } while (p != T);
    printf("(retour au debut)\n");
}
```

Fonction jozef avec expliquation

```
#include "biblio.h"

// Fonction pour créer un cercle de n noeuds, chaque noeud représentant une
position.
Node* createCircle(int n) {
    if (n <= 0) return NULL; // Si n <= 0, retourner NULL car il n'y a pas de
cercle à créer.

    // Créer le premier noeud et l'initialiser avec la position 1.
    Node* head = (Node*)malloc(sizeof(Node));
    head->position = 1;
    Node* current = head; // Pointeur pour parcourir la liste.

    // Créer les noeuds restants et les relier en cercle.
    for (int i = 2; i <= n; i++) {
        Node* newNode = (Node*)malloc(sizeof(Node)); // Allouer un nouveau noeud.
        newNode->position = i; // Assigner la position au noeud.
        current->next = newNode; // Relier le noeud actuel au nouveau noeud.
        current = newNode; // Passer au nouveau noeud.
    }

    current->next = head; // Relier le dernier noeud au premier pour former un
cercle.
    return head; // Retourner le pointeur vers le premier noeud.
}

// Fonction pour afficher le cercle.
void displayCircle(Node* head) {
    if (!head) return; // Si le cercle est vide, ne rien afficher.

    Node* temp = head; // Pointeur temporaire pour parcourir la liste.
    do {
        printf("%d ", temp->position); // Afficher la position du noeud courant.
        temp = temp->next; // Passer au noeud suivant.
    } while (temp != head); // Continuer jusqu'à revenir au premier noeud.
    printf("\n");
}

// Fonction pour résoudre le problème de Josephus.
// head : pointeur vers le premier noeud du cercle.
// k : intervalle pour éliminer les noeuds.
// start : position de départ.
int josephus(Node* head, int k, int start) {
    Node* current = head; // Pointeur pour parcourir la liste.
    Node* prev = NULL; // Pointeur pour garder une trace du noeud précédent.

    // Se déplacer jusqu'à la position de départ.
    while (current->position != start) {
        prev = current; // Mettre à jour le pointeur précédent.
        current = current->next; // Passer au noeud suivant.
    }
```

```
    // Boucle jusqu'à ce qu'il ne reste qu'un seul noeud dans le cercle.
    while (current->next != current) {
        // Avancer de k-1 positions pour trouver le noeud à éliminer.
        for (int i = 1; i < k; i++) {
            prev = current; // Mettre à jour le pointeur précédent.
            current = current->next; // Passer au noeud suivant.
        }

        // Afficher le noeud éliminé.
        printf("Eliminated: %d\n", current->position);

        // Supprimer le noeud courant et relier les noeuds restants.
        prev->next = current->next;
        free(current); // Libérer la mémoire du noeud supprimé.
        current = prev->next; // Passer au noeud suivant.
    }

    int survivor = current->position; // Récupérer la position du dernier noeud
restant.
    free(current); // Libérer la mémoire du dernier noeud.
    return survivor; // Retourner la position du survivant.
}
```

Fonction jozef sans expliquation

```
#include "biblio.h"
Node* createCircle(int n) {
    if (n <= 0) return NULL;

    Node* head = (Node*)malloc(sizeof(Node));
    head->position = 1;
    Node* current = head;

    for (int i = 2; i <= n; i++) {
        Node* newNode = (Node*)malloc(sizeof(Node));
        newNode->position = i;
        current->next = newNode;
        current = newNode;
    }

    current->next = head;
    return head;
}
```

```
void displayCircle(Node* head) {
    if (!head) return;

    Node* temp = head;
    do {
        printf("%d ", temp->position);
        temp = temp->next;
    } while (temp != head);
    printf("\n");
}



int josephus(Node* head, int k, int start) {

    Node* current = head;
    Node* prev = NULL;
    while (current->position != start) {
        prev = current;
        current = current->next;
    }

    while (current->next != current) {
        for (int i = 1; i < k; i++) {
            prev = current;
            current = current->next;
        }

        printf("Eliminated: %d\n", current->position);
        prev->next = current->next;
        free(current);
        current = prev->next;
    }



    int survivor = current->position;
    free(current);
    return survivor;
}
```

```
// Fonction pour insérer un élément dans une liste circulaire ordonnée

Cellule* inserer_ordonnee(Cellule* tete, int valeur) {
    Cellule* nouv = (Cellule*)malloc(sizeof(Cellule));
    nouv->val = valeur;

    // Cas où la liste est vide
    if (tete == NULL) {
        nouv->next = nouv; // Le seul élément pointe vers lui-même
        return nouv;
    }

    Cellule* courant = tete;
    Cellule* precedent = NULL;

    // Parcourir pour trouver la bonne position
    do {
        if (courant->val >= valeur) {
            break; // Trouver l'endroit pour insérer
        }
        precedent = courant;
        courant = courant->next;
    } while (courant != tete);

    // Insérer le nouvel élément à la bonne position
    nouv->next = courant;

    if (precedent == NULL) { // Insertion avant la tête
        Cellule* dernier = tete;
        while (dernier->next != tete) {
            dernier = dernier->next; // Trouver le dernier élément
        }
        dernier->next = nouv;
        return nouv; // Nouveau tête de la liste
    } else {
        precedent->next = nouv;
        return tete;
    }
}

// Fonction pour construire une liste circulaire ordonnée à partir d'un tableau

Cellule* construire_liste_circulaire(int tableau[], int taille) {
    Cellule* tete = NULL;

    for (int i = 0; i < taille; i++) {
        tete = inserer_ordonnee(tete, tableau[i]);
    }

    return tete;
}
```

```c
// Fonction pour afficher une liste circulaire

void afficher_liste_circulaire(Cellule* tete) {
    if (tete == NULL) {
        printf("La liste est vide.\n");
        return;
    }

    Cellule* courant = tete;
    do {
        printf("%d -> ", courant->val);
        courant = courant->next;
    } while (courant != tete);
    printf("(retour au début)\n");
}
```

```c
// Définition de la structure pour un nœud doublement chaîné

typedef struct Node {
    int val;
    struct Node* next; // Pointeur vers le nœud suivant
    struct Node* prev; // Pointeur vers le nœud précédent
} Node;
```

```c
// Fonction pour créer un nouveau nœud

Node* creer_noeud(int valeur) {
    Node* nouv = (Node*)malloc(sizeof(Node));
    nouv->val = valeur;
    nouv->next = NULL;
    nouv->prev = NULL;
    return nouv;
}
```

```
// Fonction pour ajouter un élément en fin de liste

Node* ajouter_fin(Node* tete, int valeur) {
    Node* nouv = creer_noeud(valeur);
    if (tete == NULL) {
        return nouv; // Si la liste est vide, le nouvel élément devient la tête
    }

    Node* p = tete;
    while (p->next != NULL) {
        p = p->next;
    }

    p->next = nouv; // Ajouter le nouvel élément à la fin
    nouv->prev = NULL; // Le chaînage arrière sera corrigé plus tard
    return tete;
}

// Fonction pour réaliser le chaînage arrière

void realiser_chaine_arriere(Node* tete) {
    if (tete == NULL || tete->next == NULL) {
        return; // Liste vide ou avec un seul élément, rien à faire
    }

    Node* courant = tete;
    Node* precedent = NULL;

    while (courant != NULL) {
        courant->prev = precedent; // Le pointeur prev pointe vers le nœud
précédent
        precedent = courant;       // Avancer le précédent
        courant = courant->next;   // Avancer le courant
    }
}

// Fonction pour afficher la liste dans les deux sens

void afficher_liste(Node* tete) {
    printf("Liste avant -> : ");
    Node* p = tete;
    Node* dernier = NULL;

    while (p != NULL) {
        printf("%d -> ", p->val);
        dernier = p;
        p = p->next;
    }
    printf("NULL\n");

    printf("Liste arrière <- : ");
    while (dernier != NULL) {
        printf("%d -> ", dernier->val);
        dernier = dernier->prev;
    }
    printf("NULL\n");
}
```

```c
typedef struct cel {
    int val;
    struct cel* next;
} cel;

// Fonction pour créer une pile vide

cel* creer_pile() {
    return NULL; // Une pile vide est représentée par un pointeur NULL
}

// Fonction pour empiler (ajouter un élément à la pile)

cel* empiler(cel* sommet, int valeur) {
    cel* nouv = (cel*)malloc(sizeof(cel));
    nouv->val = valeur;
    nouv->next = sommet; // Le nouveau nœud pointe vers l'ancien sommet
    return nouv;         // Retourne le nouveau sommet
}

// Fonction pour dépiler (retirer un élément de la pile)

cel* depiler(cel* sommet, int* valeur) {
    if (sommet == NULL) {
        printf("Pile vide !\n");
        *valeur = -1; // Indique une erreur
        return NULL;
    }
    cel* temp = sommet;
    *valeur = temp->val;
    sommet = temp->next; // Le sommet devient l'élément suivant
    free(temp);          // Libère la mémoire de l'ancien sommet
    return sommet;
}

// Fonction pour afficher la pile


void afficher_pile(cel* sommet) {
    printf("Pile : ");
    while (sommet != NULL) {
        printf("%d -> ", sommet->val);
        sommet = sommet->next;
    }
    printf("NULL\n");
}
```

```c
// Structure pour la file (avec front et rear pour gérer les extrémités)

typedef struct file {
    cel* front; // Début de la file
    cel* rear;  // Fin de la file
} file;

// Fonction pour créer une file vide

file* creer_file() {
    file* nouvelle_file = (file*)malloc(sizeof(file));
    nouvelle_file->front = nouvelle_file->rear = NULL; // Initialement vide
    return nouvelle_file;
}

// Fonction pour enfiler (ajouter un élément à la fin de la file)

void enfiler(file* f, int valeur) {
    cel* nouv = (cel*)malloc(sizeof(cel));
    nouv->val = valeur;
    nouv->next = NULL;

    if (f->rear == NULL) { // Si la file est vide
        f->front = f->rear = nouv;
        return;
    }

    f->rear->next = nouv; // Ajoute l'élément à la fin
    f->rear = nouv;       // Met à jour le pointeur rear
}

// Fonction pour défiler (retirer un élément du début de la file)

int defiler(file* f) {
    if (f->front == NULL) {
        printf("File vide !\n");
        return -1; // Indique une erreur
    }

    cel* temp = f->front;
    int valeur = temp->val;
    f->front = f->front->next;

    if (f->front == NULL) { // Si la file devient vide
        f->rear = NULL;
    }

    free(temp); // Libère la mémoire de l'ancien nœud
    return valeur;
}
```

```
// Fonction pour afficher la file

void afficher_file(file* f) {
    cel* courant = f->front;
    printf("File : ");
    while (courant != NULL) {
        printf("%d -> ", courant->val);
        courant = courant->next;
    }
    printf("NULL\n");
}


// Définition de la structure pour un nœud de pile

typedef struct Node {
    int val;
    struct Node* next;
} Node;

// Définition de la pile

typedef struct {
    Node* top; // Sommet de la pile
} Pile;

// Fonction pour créer une nouvelle pile vide

Pile* nouvellePile() {
    Pile* pile = (Pile*)malloc(sizeof(Pile));
    pile->top = NULL;
    return pile;
}

// Fonction pour vérifier si la pile est vide

int estPileVide(Pile* pile) {
    return (pile->top == NULL);
}

// Fonction pour empiler une valeur dans la pile

void empiler(Pile* pile, int valeur) {
    Node* nouv = (Node*)malloc(sizeof(Node));
    nouv->val = valeur;
    nouv->next = pile->top;
    pile->top = nouv;
}
```

```
// Fonction pour dépiler une valeur de la pile

int depiler(Pile* pile) {
    if (estPileVide(pile)) {
        printf("Erreur : Pile vide !\n");
        return -1; // Indicateur d'erreur
    }

    Node* temp = pile->top;
    int valeur = temp->val;
    pile->top = temp->next;
    free(temp);
    return valeur;
}
```

Au lieu d'appeler une fonction récursive, chaque appel récursif est simulé en ajoutant une valeur à la pile.

- Par exemple, pour n=5n = 5n=5, nous ajoutons 5,4,3,2,15, 4, 3, 2, 15,4,3,2,1 à la pile.

```
int somme_simulee(int n) {
    Pile* pile = nouvellePile(); // Créer une pile
    int somme = 0;

    // Empiler tous les états simulant la récursivité
    for (int i = n; i > 0; i--) {
        empiler(pile, i);
    }

    // Dépiler et calculer la somme
    while (!estPileVide(pile)) {
        somme += depiler(pile);
    }

    free(pile); // Libérer la mémoire de la pile
    return somme;
}
```