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;
}
```

```
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

```
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

```
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");
}
```