Ce corrigé a été fait hors le temps imparti ! Mon interprétation des consignes a sensiblement évolué au cours du temps.

Partie I : Implémentation naïve

Dans cette partie, on s'autorise à agir directement sur la liste `mem`. Toutefois un usage dévoyé des fonctions `lire` et `ecrire` était possible (en commentaire).

Question 1. On applique la consigne en supposant que les cases sont disponibles dans la liste `mem`.

```
def initialiser(p, n, c): #cette fonction agit par effet de bord
    for i in range(n): # et renvoie None par conséquent
        mem[p+i] = c #ecrire(p, i, c)
```

Question 2. La liste `mem` venant d'être initialisée, la première portion disponible est donc située à l'indice `1`. La fonction agit par effet de bord.

```
def demarrage():
    mem[0] = 1 #ecrire(0, 0, 1)
```

Question 3. La condition pour pouvoir réserver les nouvelles cases est `prochain+n-1<=TAILLE_MEM-1` si cette condition n'est pas réalisée la fonction ne fait rien et renvoie `None` comme voulu.
Il est préférable de renvoyer `None` de façon explicite surtout que c'est demandé.

```
def reserver(n, c):
    p = mem[0] #=prochain=lire(0,0)
    if p + n<=TAILLE_MEM:
        initialiser(p, n, c)
        mem[0]+= n
        return p
    else:
        return None
```

L'exécution de `initialiser(p, n, c)` est clairement en $\mathcal{O}(n)$. Cette exécution est appelée lors de l'exécution de `reserver(n, c)` à laquelle s'ajoute quelques opérations qui s'effectuent en temps constant.
Ainsi la complexité de `reserver(n, c)` est en $\mathcal{O}(n)$.

Partie II : Réservations de blocs de tailles fixes

À partir de maintenant, on s'interdit absolument l'accès direct à la liste `mem` et je privilégie les fonctions introduite le plus tardivement possible. Exemple en question 4, je préfère `ecrire_prochain(2)` à `ecrire(0, 0, 2)`.

Question 4. La liste `mem` étant initialisée (liste constituée uniquement de 0), tout les blocs sont donc libres ainsi la première position libres est celle d'indice 2.

```
def demarrage():
    ecrire_prochain(2)
```

Question 5. La première fonction recherche la première position éventuellement disponible.

```
def premier_bloc_libre():# On commence donc par rechercher le premier bloc libre, s'il existe.
    p = 2
    prochain = lire_prochain()
    while est_reservee(p) and p < prochain:
        p+=TAILLE_BLOC
    return [p, prochain]

def reserver(n, c):
    if n >= TAILLE_BLOC: # bloc trop petit
        return None
    [p, prochain] = premier_bloc_libre()
    if p+n >= TAILLE_MEM:#mémoire pleine
        return None
    initialiser(p, n, c)
    marque_reservee(p)
    if p >= prochain:
        ecrire_prochain(p+TAILLE_BLOC)
    return p
```

Il y a au plus `TAILLE_MEM/TAILLE_BLOC` tours de boucles dans la fonction `premier_bloc_libre` et l'appel de `initialiser(p, n, c)` a une complexité en $\mathcal{O}(n)$ (vu en partie I), les autres opérations s'effectuent en temps constant. La complexité de `reserver(n, c)` est donc en $\mathcal{O}(\texttt{n + TAILLE\_MEM}) = \mathcal{O}(\texttt{TAILLE\_MEM})$.

En poussant le bouchon : si `n>=TAILLE_BLOC` la fonction s'effectue en temps constant. Si `0<n<TAILLE_BLOC` alors on a `TAILLE_MEM/TAILLE_BLOC<= TAILLE_MEM/n`. La complexité de `reserver` est donc aussi en $\mathcal{O}(\texttt{n + TAILLE\_MEM/n})$. Finalement la complexité de `reserver` est en $\mathcal{O}(\max(\texttt{n + TAILLE\_MEM/n, TAILLE\_MEM}))$ *Je ne pense pas que cela soit la réponse attendue mais cela respecte la consigne !*

Question 6. A priori, on libère le bloc sans se poser trop de questions.

```
def liberer(p) :
    marque_libre(p)
```

Vu la stratégie, on ne gagnerait pas à mettre à jour « `prochain` » (comme ci-dessous), en le faisant pointer vers la première portion libre qui succède à la dernière portion réservée. Cela ne changera pas la complexité des autres fonctions (les précédentes) qui gèrent la mémoire. L'intérêt de cette deuxième version est que l'on pourra avoir une meilleure estimation de la mémoire disponible en calculant :

`TAILLE_MEM-lire_prochain()-(TAILLE_MEM-lire_prochain())//TAILLE_BLOC`

```
def liberer(p) :
    marque_libre(p)
    if p+TAILLE_BLOC == lire_prochain():
        q=p
        while est_libre(q) and q > 2:
            q-=TAILLE_BLOC
        ecrire_prochain(q+TAILLE_BLOC)
```

Partie III : Portions avec en-tête et pied de page

Question 7. L'entier encodant l'information dans les cases additionnelles d'un bloc est de la forme `mot=taille+r` où `taille` (qui vaut n ou $n+1$) est le nombre pair de cases réservées pour la portion et où `r` vaut `0` si le bloc est libre et vaut `1` sinon.

`r` est donc le reste de la division euclidienne de `mot` par 2, la taille est elle égale à deux fois le quotient. En effet on peut écrire `mot=2*q+r` où `q` est le quotient.

L'information concernant une portion `p` est stockée dans `mem` juste avant et après celles-ci, aux indices `p-1` et `p+taille` et les données (de type caractères) sont stockées dans la portion `p` aux indices : `p`, `p+1`,..., `p+taille-1`.

L'information concernant l'éventuelle portion située juste avant la portion `p` est stockée en fin de ce bloc et se trouve donc en indice `p-2`.

`est_reservee(p)` renvoie ainsi un booléen (`r==1`) indiquant si la portion `p` est réservée ;

`marque_reservee(p, taille)` met à jour les cases deux cases d'indices `p-1` et `p+taille` contenant l'information sur la portion `p` pour la marquer comme réservée ;

`precedent_est_libre(p)` renvoie un booléen indiquant si la portion qui précède `p` est libre.

Question 8. Pour `demarrage()` que l'on applique alors que la mémoire est remplie de 0, on initialise les portions épilogue et prologue ainsi que la position de l'épilogue.

```
def demarrage():
    ecrire_position_epilogue(PROLOGUE + 2)
    marque_reservee(PROLOGUE, 0)
    marque_reservee(PROLOGUE + 2, 0)#portion épilogue
```

Question 9. On commence par chercher une portion libre de taille paire $\geqslant n$ (n ou $n+1$ suivant la parité de n).

```
def reserver(n, c):# on commence par chercher une portion libre de taille>n-1
    taille= n + n%2 # taille de réservation="entier pair>=n"
    EPILOGUE = lire_position_epilogue()
    pc=PROLOGUE+2 #position courante à tester
    t_pc=lire_taille(pc) #taille position courante
    while pc < EPILOGUE and (est_reservee(pc) or t_pc<taille):
        pc+= t_pc + 2
        t_pc=lire_taille(pc)
    if pc<EPILOGUE:#pc>=EPILOGUE ssi aucune portion ne convient
        if t_pc > taille+3:# On crée une portion libre de taille non nulle
            marque_libre(pc+2+taille, t_pc-taille-2)
        else: taille=tpc #on va prendre toute la portion
    elif TAILLE_MEM > taille + 2 + EPILOGUE:
        ecrire_position_epilogue(pc+taille+2)
        marque_reservee(pc+taille+2,0)#on déplace l'épilogue
    else: return None #pas de places de réservations avec la stratégie
    marque_reservee(pc, taille)
    initialiser(pc, n, c)
    return pc
```

Les fonctions `lire_taille`, `lire_position_epilogue`, `est_reservee`, `marque_libre`, `ecrire_position_epilogue`, `marque_libre` et `marque_reservee` s'effectuent en temps constant. L'exécution de `initialiser(pc, n, c)` est en $\mathcal{O}$(`n`) mais cette exécution est effectuée uniquement si `n` $<$ `TAILLE_MEM`. Le nombre de tour de boucles inconditionnelles (`while`) est en $\mathcal{O}$ (`TAILLE_MEM`). La fonction `reserver` est donc en $\mathcal{O}$ (`TAILLE_MEM`). En faisant comme en 5. (*technique de poussage de bouchons !*), on aurait une complexité en $\mathcal{O}$ (max (`n + TAILLE_MEM/n, TAILLE_MEM`)).

Question 10. On remarque qu'aucune des fonctions qui gèrent la mémoire ne créent pas de blocs libres adjacents, il suffit lors de la libération d'un bloc de fusionner celui-ci avec les éventuels blocs libres parmi les deux qui lui sont adjacents.

Il y a donc au plus deux fusions à chaque appel de la fonction `liberer`.

De sorte que l'invariant « *il n'y a pas de portions adjacentes libres* » sera conservé. Cela assure la correction de la fonction.

Par conséquent, si on libère un bloc il suffit de vérifier si le bloc prédécesseur ou le bloc successeur est libre. L'intérêt des portions prologue et épilogue est qu'elles sont marquées occupées cela permet d'éviter les erreurs et de ne pas tenter de fusion malencontreuse.

```
def liberer(p):
    pl=p#portion à libérer
    tl=lire_taille(pl)#taille bloc à libérer
    pd=p+tl+2#portion de droite
    if est_libre(pd):# si possible à droite
        td=lire_taille(pd)#taille bloc de droite
        tl+=td+2#la taille mise à jour pour fusion droite
    if precedent_est_libre(pl):#test à gauche
        tg=lire_taille_precedent(pl)#taille gauche
        pl=pl-tg-2#portion mise à jour pour fusion à  gauche
        tl+=tg+2#taille mise à jour pour fusion à  gauche
    marque_libre(pl, tl)
```

Il n'y a pas de boucle et toutes les fonctions et opérations exécutées par `liberer` s'effectuent en temps constant, la fonction `liberer` a une complexité en : $\mathcal{O}(1)$.

Partie IV : Chaînage explicite des portions libres

Question 11.

```
def ajout_en_entree_de_chaine(p):
    np=lire_entree_chaine()
    ecrire_entree_chaine(p)
    ecrire_predecesseur(p,0)
    ecrire_successeur(p,np)
    if np!=0: ecrire_predecesseur(np,p)
```

Question 12. Je propose le code suivant qui est valable même si `suc` ou `pre` valent `0`.

```
def supprime_dans_chaine(p):
    pre=lire_predecesseur(p)
    suc=lire_successeur(p)
    ecrire_successeur(pre,suc)
    if suc!=0: ecrire_predecesseur(suc,pre)
```

Question 13. On reprend l'idée de III, en se servant du fait qu'à l'initialisation de `mem`, il n'y a que des `0`.

```
def demarrage():
    ecrire_position_epilogue(PROLOGUE+2)
    marque_reservee(PROLOGUE, 0)
    marque_reservee(PROLOGUE+2, 0)
```

Question 14. On effectue les mêmes opérations mais on parcourt la chaîne des portions libres.

Je note donc `npl` le nombre de portion libres. A priori on a `npl`$\ll$`TAILLE_MEM`.

```
def reserver(n, c):
    taille= n + n%2 # taille de réservation="entier pair>=n"
    pc = lire_entree_chaine()
    t_pc = lire_taille(pc)# renvoie n'importe quoi si pc==0
    while pc!=0 and t_pc<taille:
        pc = lire_successeur(pc)
        t_pc = lire_taille(pc)
    if pc!=0:
        if t_pc < taille+3:
            taille=tpc
        else:
            marque_libre(pc+2+taille, t_pc-taille-2)
            ajout_en_entree_de_chaine(pc+2+taille)
    elif TAILLE_MEM > taille + 2 + EPILOGUE:
#Les blocs libres sont trop petits et on a la place en mémoire
# donc on déplace l'épilogue
        pc = lire_position_epilogue()
        ecrire_position_epilogue(pc+taille+2)
        marque_reservee(pc+taille+2,0)
    else: #pas de place dans MEM
        return None
    marque_reservee(pc, taille)
    initialiser(pc, n, c)
    return pc
```

La complexité est donc en $\mathcal{O}(\mathtt{n}+\mathtt{npl})$.

Remarque : les "0" des cases 23 et 24 de la figure 8 ne correspondent pas aux consignes et sont sources d'erreurs.

Question 15. Par rapport à la fonction `liberer` de la partie précédente, il s'agit de mettre à jour la chaîne des portions libres. Ces fonctions s'effectuant en temps constant et comme il y en a au plus 2 portions à fusionner. La fonction `liberer` est encore en $\mathcal{O}(1)$.

```
def liberer(p):
    pl=p#portion à libérer
    tl=lire_taille(pl)#taille bloc à libérer
    pd=p+tl+2#position du bloc de droite
    if est_libre(pd):# si possible à droite
        td=lire_taille(pd)#taille bloc de droite
        tl+=td+2#la taille mise à jour pour fusion droite
        supprime_dans_chaine(pd)
    if precedent_est_libre(pl):#test à gauche
        tg=lire_taille_precedent(pl)#taille gauche
        pl=pl-tg-2#portion mise à jour pour fusion à  gauche
        tl+=tg+2#taille mise à jour pour fusion à  gauche
        supprime_dans_chaine(pl)
    ajout_en_entree_de_chaine(pl)
    marque_libre(pl, tl)
```