1. Primitives pour les listes chainees.

```
InsererApres (L, P, x) {
   new (Q);
   Q.info <- x;
   Q.suivant <- P.suivant;
   P.suivant <- Q;
}

InsererEnTete (L, x) {
   new (P);
   P.info <- x;
   P.suivant <- L;
   L <- P;
}

SupprimerApres (L, P) {
   P.suivant <- P.suivant.suivant;
}

SupprimerEnTete (L) {
   L <- L.suivant;
}
```

2. Algorithmes sur les listes chainees.

```
ListeInverser (L) {
   Linv <- nil;
   P <- L;
   Tant que (P != nil) Faire {
      InsererEnTete (Linv, P.info);
      P <- P.suivant;
   }
   retourner Linv;
}
```

```
ListeDupliquer (L) {
   Lcp <- nil;
   P <- L;
   Si (P != nil) Alors {
      InsererEnTete (Lcp, P.info);
      Pcp <- Lcp;
      P <- P.suivant;
   }
   Tant que (P != nil) Faire {
      InsererApres (Lcp, Pcp, P.info);
      Pcp <- Pcp.suivant;
      P <- P.suivant;
   }
   retourner Lcp;
}

ListeConcatener (L1, L2) {  // Creation d'une nouvelle liste
   L1cp <- ListeDupliquer (L1);
   L2cp <- ListeDupliquer (L2);

   Si (L1cp = nil) Alors
      retourner L2cp;

   P <- L1cp;
   Tant que (P.suivant != nil) Faire
      P <- P.suivant;
   P.suivant <- L2cp;
   retourner L1cp;
}

ListeSupprimer (L, Predicat) { // Version 1
   Tant que (L != nil ET Predicat (L.info)) Faire
      SupprimerEnTete (L);

   P <- L;
   Si (P != nil) Alors {
      Tant que (P.suivant != nil) Faire {
         Si Predicat (P.suivant.info) Alors
            SupprimerApres (L, P);
         Sinon
            P <- P.suivant;
      }
   }
}

ListeSupprimer (L, Predicat) { // Version 2
   P <- L;
```

```
   P_prec <- nil;

   Tant que (P != nil) Faire {
      Si Predicat (P.info) Alors {
         Si (P_prec = nil) Alors {
            SupprimerEnTete (L)l
            P <- L;
         }
         Sinon {
            SupprimerApres (L, P_prec);
            P <- P_prec.suivant;
         }
      }
      Sinon {
         P_prec <- P;
         P <- P.suivant;
      }
   }
}

ListeTrieeInserer (L, x) { // Version 1
   Si (L = nil OU x <= L.info) Alors
      InsererEnTete (L, x);
   Sinon {
      P <- L;
      Tant que (P.suivant != nil ET P.suivant.info < x) Faire
         P <- P.suivant;
      InsererApres (L, P, x);
   }
}

ListeTrieeInserer (L, x) { // Version 2
   P <- L;
   P_prec <- nil;
   Tant que (P != nil ET P.info < x) Faire {
      P_prec <- P;
      P <- P.suivant;
   }
   Si (P_prec = nil) Alors
      InsererEnTete (L, x);
   Sinon
      InsererApres (L, P_prec, x);
}


// Fonction de recherche renvoyant une paire (booleen, pointeur)
// Si l'element x appartient a la liste, le booleen est VRAI et le pointeur
```

```
// indique la position de l'element dans la liste (pointeur vers le precedent).
// Si l'element x n'appartient pas a la liste, le booleen est FAUX et le
// pointeur indique la position d'insertion de l'element (pointeur vers le
// precedent).

ListeTrieeRecherche (L, x) {
   Si (L = nil OU x < L.info) Alors
      Retourner (FAUX, nil);

   Si (x = L.info) Alors
      Retourner (VRAI, nil);

   P <- L;
   Tant que (P.suivant != nil ET P.suivant.info < x) Faire
      P <- P.suivant;
    Si (P.suivant = nil OU P.suivant.info > x) Alors
       retourner (FAUX, P);
    retourner (VRAI, P);
}

ListeTrieeInserer (L, x) { // Version 3
   (est_element, P) <- ListeTrieeRecherche (L, x);
   Si (P = nil) Alors
      InsererEnTete (L, x);
   Sinon
      InsererApres (L, P, x);
} // Si on souhaite eviter les doublons, il faut tester est_element

ListeTrieeSupprimer (L, x) {
   (est_element, P) <- ListeTrieeRecherche (L, x);
   Si (est_element) Alors {
      Si (P = nil) Alors
         SupprimerEnTete (L);
      Sinon
         SupprimerApres (L, P);
   }
}
```