University of Setif 1
Faculty of Science
Department of Computer Science
Major: Academic Bachelor's degree
Module: Algorithmics and Data Structure

Academic year: 2022/23/24

TD/TP N° 2

Linked Linear Lists

Exercise 1 : - Operations (functions) on simply chained Linear lists

We are now proposing a new chained list structure, which is:
Track the **list of products purchased** by a customer at a convenience store

Each item in the list consists of the following fields:
code_prod(9 alphanumeric),
Designation(50 car),
UM (3 cars), // Unit of measurement (U, Kg, L, end, cart, ...)
PUA_HT(actual), // unit purchase price excluding tax
Qty(actual), // Quantity purchased
VAT (9% or 21%) // Value Added Tax

In tutorials: Writing a program/Interactive algorithm for managing the list of purchased products

This program will display the following menu:

1. ADD Products At the top of the list
2. VIEW the list of products purchased
3. Display the total number of products purchased as well as the total amounts excl. VAT, total VAT and total TTC (net to be paid)
4. Ask the customer for payment and calculate the difference between the **amount paid** and the **net amount to be paid**
5. DELETE products
6. Show max and min price
7. EMPTY the list.

8. STOPPING the program.

And will carry out the processing (functions/procedures) corresponding to the choice made.

Bonus: If the profit is 10% for all products, calculate the Total profit

In practical work, translate into C/C++ the algorithms developed in TD

Exercise 2: Backchaining a Doubly Linked List

Design an algorithm that performs backchaining of a double-chained list that has only fronted chaining.

Exercise 3: Insert and delete in a double-linked list

- Write an algorithm for inserting into a double-linked list.
- Write a deletion algorithm in a double-linked list.

Exercise Answers

Exercise 1: Managing the products purchased by a customer in a supermarket

See course with similar examples

Exercise 2: Performing the back-chaining of a double-linked list

Spécification de l'algorithme

```
FONCTION doubleChain(l : list)
VAR tête : list
DEBUT
 SI (l = NULL) ALORS
         RETOURNER // ceci termine la fonction
 FINSI
 tête ← l
 TANTQUE (l →succ ≠ NULL) ET (l →succ ≠ tête) FAIRE
         SI (l →succ→prev = NULL) ALORS
                 l →succ→prev ← l
         FINSI
                 l ← l →succ
 FINTQ
         SI (l →succ ≠ NULL) ALORS
                 tête→prev ← l
         FINSI
         RETOURNER
FIN
```

Implantation C

```c
void doubleChain(list l)
{
 if (l == NULL) return;
 list head = l;
 while (l->succ != NULL && l->succ != head)
  {
   if (l->succ->prev == NULL) l->succ->prev = l;
   l = l->succ;
  }
 if (l->succ != NULL) head->prev = l;
}
```

Exercise 3: Insert and delete in a double-linked list

Explanation:

We proceed in the same way as for the case of a simply linked list, except that the algorithm is simpler. Indeed, the problem for a simply linked list consists in identifying the direct predecessor of the insertion or elimination point, and therefore implies a linear path starting from the head.

In the case of a double-linked list, you have direct access to the previous element in the chain, thus eliminating this difficulty. For the rest, it is a question of updating the predecessors in addition to the successors.

The problems of insertion/deletion at the beginning or end of lists, vs. in the middle of the list remain more or less the same as with a simply linked list.

Insertion

```
Spécification de l'algorithme
// Insertion dans une liste doublement chaînée
// pl : liste où s'effectue l'insertion
// place : place dans la liste (on insère juste devant)
// k : élément à insérer
// status code : 0 : ok, -1 : non ok
// Type abstrait : list inserer(list l, int k, element e);

FONCTION insert(*pl : list, place :list , k : entier) : booléen
VAR noeudk, last, prec : list
DEBUT
 // la place concernée est-elle bien dans la liste ?
 SI ¬ appartient(place, *pl) ALORS
  RETOURNER faux
 FINSI
 noeudk ← newSingletionList(k) // création d'un noeud
 // cas de la liste vide
 SI *pl= NULL ALORS
  *pl ← noeudk
  RETOURNER vrai
 FINSI
 // cas de la place en tête de liste
 SI place = *pl ALORS // noeudk mis en tête de liste
  noeudk→succ ← *pl
  noeudk→prec ← NULL
  *pl ← noeudk
  RETOURNER vrai
 FINSI
 // cas de la place en fin de liste
 SI place = NULL ALORS
  last ← lastElement(*pl) // on trouve le dernier élément de la liste
  last→succ ← noeudk
  noeudk→prev ← last
  RETOURNER vrai
 FINSI
 // autre cas : place en milieu de liste
 prec ← place→prev
 prec→succ ← noeudk
 noeudk→succ ← place
 noeud→kprev ← prec
 place→prev ← noeudk
 RETOURNER vrai
FIN
```

```c
Implantation C
// Insertion dans une liste doublement chaînée
// pl : liste où s'effectue l'insertion
// place : place dans la liste (on insère juste devant)
// k : élément à insérer
// status code : 0 : ok, -1 : ko
// Type abstrait : list inserer(list l, int k, element e);
int insert(list* pl, list place, int k)
{
 // vérifier que la place est bien dans la liste concernée
 if (!(areConvergent(place, *pl))) return -1;
 list K = newSingleton(k);
 // cas de la liste vide
 if (*pl == NULL)
```

```
  {
    *pl = K;
    return 0;
  }
 // cas tête de liste :
 if (place == *pl)
  {
    K->succ = *pl;
    K->prev = NULL;
    *pl = K;
    return 0;
  }
 // cas fin de liste :
 if (place == NULL)
  {
    list last = lastElement(*pl);
    last->succ = K;
    K->prev = last;
    return 0;
  }
 // sinon, milieu de liste
 list prec = place->prev;
 prec->succ = K;
 K->succ = place;
 K->prev = prec;
 place->prev = K;
 return 0;
}
```

Deletion

Spécification de l'algorithme

```
// Suppression dans une liste doublement chaînée
// Type abstrait : list supprimer(list l, int k);
// status code : 0 : ok, -1 : ko

FONCTION removeElement(*pl :list, place : list) : booléen
VAR prec : list
DEBUT
 SI ¬ appartient(place, *pl) ALORS
  RETOURNER faux
 FINSI
 // cas de la liste vide
 SI isempty(*pl) ALORS // autre moyen de tester la liste vide
  RETOURNER vrai
 FINSI
 // cas de la place en tête de liste
 SI place = *pl ALORS
  *pl ← (*pl)→succ
  SI place→succ ≠ NULL ALORS
   (*pl)→succ→prev ← NULL
  FINSI
  LIBERER(place)
  RETOURNER vrai
 FINSI
 prec ← place→prev
 prec→succ ← place→succ
```

```
  SI place→succ ≠ NULL ALORS
    place→succ→prev ←  prec
  FINSI
  LIBERER(place)
  RETOURNER vrai
FIN
```

Implantation C

```c
// Deleting from a Doubly Linked List
// Type abstrait : list supprimer(list l, int k);
// status code : 0 : ok, -1 : ko
int removeElement(list* pl, list place)
{
// vérifier que la place est bien dans la liste concernée
if (!(areConvergent(place, *pl))) return -1;
// cas de la liste vide
if (*pl == NULL) return 0;
// cas tête de liste :
if (place == *pl)
{
*pl = (*pl)->succ;
if (place->succ != NULL) (*pl)->succ->prev = NULL;
free(place);
return 0;
}
list prec = place->prev;
prec->succ = place->succ;
if (place->succ != NULL) place->succ->prev = prec;
free(place);
return 0;
}
```