Niveau: 1e année informatique
Matière: ASD2

Corrige type TD/TP N° : 04

Année universitaire : 2022/2023
Chapitre 1 : Les listes chainées

Remarque utiliser l'opérateur **new** et **delete** (C++)

Exercice 1 : (TD/TP)

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Cell {
  int data;
  struct Cell* next;
} Cell;

typedef struct List {
  int size;
  Cell* head;
} List;

List initList() {
  List lst;
  Lst.size=0;
  Lst.head=NULL;
  return lst;
}

bool add_head(List &lst, int d) {
  Cell* e = new Cell ;
  if (e == NULL)
    return false;
  e->data = d;
  e->next = lst.head;
  lst.head=e;
  lst.size++;
  return true;
}

bool append_end(List &lst, int d) {
  Cell* t;
  Cell* e = new Cell ;
  if (e == NULL)
    return false;
  e->data = d;
  e->next = NULL;
  if (lst.head == NULL)
    lst.head = e;
  else {
    t= lst.head;
    while (t->next != NULL)
      t= t->next;
    t->next=e;
  }
  Lst.size++;
  return true;
}

bool delete_head(List &lst) {
  Cell* t;
  if (  lst.head== NULL)
    return false;
  t = lst.head;
  lst.head =t->next;
  free(t);
  lst.size--;
  return true;
}

bool delete_end (List &lst) {
  Cell* t, *p;
  if (lst.head== NULL)
    return false;
  t = lst.head;
  if (t->next ==NULL) {
    free(lst.head);
    lst.head = NULL;
  } else {
    while (t->next != NULL) {
```

```
      p=t;
      t= t->next;
    }
    p->next=NULL;
    free(t);
  }
  Lst.size--;
  return true;
}

void display_list(List lst) {
  Cell* h=lst.head;
  while (h != NULL) {
    printf("%d->", h->data);
    h = h->next;
  }
  printf("fin\n");
}

int main() {
  List l=initList();
  display_list(l);
  add_head(l,5);
  display_list(l);
  append_end(l,7);
  display_list(l);
  add_head(l,3);
  display_list(l);
  delete_end(l);
  display_list(l);
  delete_head(l);
  display_list(l);
  return 0;
}
```

Exercice 2 : (TP)

Écrivez la fonction « **get »** qui renvoie la valeur de l'élément situé dans un emplacement, et si l'emplacement n'existe pas, il quitte le programme. « **exit**(-1) »

```
int get (List h,int pos) {
  int i=1;
  if(pos<1) exit(-1);
  while (h != NULL) {
    if(i==pos) return h->data;
    h = h->next;
    i++;
  }
  exit(-1);
}

int main() {
  List head =NULL;
  int p=2;
  add_head(head, 3);
  add_head(head, 2);
  append_end(head, 4);
  add_head(head, 1);
  append_end(head, 5);
  printf("size=%d\n", size_list(head));
  display_list(head);
  printf("list(%d)=%d\n", p, get(head,p));
  return 0;
}
```

Exercice 3 : (TD)

Écrivez la fonction « **insert** » qui ajoute un élément à la liste à l'emplacement spécifié, et si l'emplacement est inférieur à 1, elle l'ajoute au début de la liste, mais s'il est supérieur à la taille de la liste, elle l'ajoute à la fin de la liste.

```
bool insert(List &Head, int d,int pos) {
  if (pos<=1) return add_head(Head,d);
  if (pos>size_list(Head)) return append_end(Head,d);
  List t;
  List e = new Node;
  if (e == NULL)
    return false;
  e->data = d;
  t=Head;
  for (int i=1;i<pos;i++)
    t = t->next;
  e->next = t->next;
  t->next = e;
  return true;
}

bool insert2(List& Head, int d, int pos) {
  List t, p, e= new Node;
  e->data = d;
  if ((Head == NULL)(pos<=1) ) {
    e-> next = NULL;
    (Head) = e;
  } else {
    t = (Head);
    for (int i=1; i<pos && t->next; i++)
      t=t->next;
    e->next=t->next;
    t->next=e;
  }
}


int main() {
  List head =NULL;
  int p=-12;
  add_head(head, 3);
  add_head(head, 2);
  append_end(head, 4);
  add_head(head, 1);
  append_end(head, 5);
  printf("size=%d\n", size_list(head));
  display_list(head);
  insert(head, 22,p);
  display_list(head);
  return 0;
}
```

Exercice 4 : (TP)

Écrivez la fonction **tab2list** qui remplit une liste à partir d'un tableau.

```
List tab2list(int *t, int n) {
  List l =NULL;
  for (int i=0; i<n; i++)
    append_end(l,t[i]);
  return l;
}

int main() {
  int t[]={1,2,3,4,5,6};
  List h =tab2list(t, 6);
  display_list(h);
  return 0;
}
```

Exercice 5 : (TD)

Écrivez la fonction **contains** qui nous indique si un nombre est dans la liste ou non.

```
bool contains (List h, int d) {
  while (h != NULL) {
    if(h->data==d) return true;
    h = h->next;
  }
```

```
  return false;
}
```

Exercice 6 : (TP)

Écrivez la fonction **insertSorted** qui ajoute un élément à une **liste triée** par ordre croissant tout en maintenant l'ordre de tri.

```
void insertSorted(List &Head, int x) {
  List t, e = new Node;
  e->data = x;
  e->next = NULL;
  if (Head == NULL || x < (*aHead)->data) {
    e->next = Head;
    Head = e;
  } else {
    t = Head;
    while (t->next!=NULL && t->next->data < x)
      t = t->next;
    e->next = t->next;
    t->next = e;
  }
}

int main() {
  List h =NULL;
  insertSorted(h, 5);
  insertSorted(h, 9);
  insertSorted(h, 7);
  insertSorted(h, 3);
  display_list(h);
  return 0;
}
```

Exercice 7 : (TD)

Ecrire la fonction **inverseList** qui inverse une liste

```
void inverseList(List &Head) {
  List p, t=Head;
  Head =NULL;
  while(t) {
    p=t;
    t=t->next;
    p->next=*aHead;
    Head =p;
  }
}

int main() {
  List h =NULL;
  insertSorted(h, 5);
  insertSorted(h, 9);
  insertSorted(h, 7);
  insertSorted(h, 3);
  display_list(h);
  inverseList(h);
  display_list(h);
  return 0;
}
```

Exercice 8 : (TD/TP)

Écrivez la fonction **supDouble** qui supprime tous les doublons d'une **file triée** et ne laisse qu'une seule occurrence de chaque élément.

```
typedef struct Queue {
  struct Node* first, *last;
  int size;
} Queue;

Queue * initQ() {
  Queue *Q=new Queue;
```

```
  Q->first =NULL;
  Q->last =NULL;
  Q->size =0;
}

bool isEmpty(Queue* Q) {
  return Q->size ==0;
}

void enQueue(Queue *Q, int x) {
  Node *e = new Node;
  e->data = x;
  e->next = NULL;
  if(isEmpty(Q))
    Q->first =e;
  else
    Q->last->next =e;
  Q->last =e;
  Q->size++;
}

int deQueue(Queue *Q) {
  Node* t;
  int x;
  if(isEmpty(Q)) {
    printf("error: Queue is empty");
    exit(1);
  }
  t = Q->first;
  x = t->data;
  Q->first = t-> next;
  free(t);
  Q->size--;
  return x;
}

void supDouble(Queue *Q) {
  int x,y,s,i;
  s=Q->size;
  x=deQueue(Q);
  y=x;
  for(i=1;i<s;i++) {
    y=deQueue(Q);
    if(x!=y) {
      enQueue(Q,x);
      x=y;
    }
  }
  enQueue(Q,x);
}

int main() {
  Queue *Q=initQ();
  enQueue(Q, 1);
  enQueue(Q, 1);
  enQueue(Q, 1);
  enQueue(Q, 2);
  enQueue(Q, 2);
  enQueue(Q, 3);
  enQueue(Q, 4);
  enQueue(Q, 4);
  enQueue(Q, 4);
  enQueue(Q, 5);
  display_list(Q->first);
  supDouble(Q);
  display_list(Q->first);
  return 0;
}
```

Exercice 9 : (TD/TP)

Écrivez la fonction **inverseTable** qui inverse une table à l'aide d'une **pile**.

```
int Pop(List& Head) {
  List t;
  int x;
  if(Head ==NULL) {
    printf("error: Stack is empty");
    exit(1);
  }
  t = Head;
  *aHead =t->next;
  x=t->data;
  free(t);
  return x;
}

void Push(List& Head, int x) {
  List e = new Node;
  e-> data = x;
  e-> next = Head;
  Head =e;
}

void inverseTable(int*tab, int size) {
  List lst=NULL;
  for(int i=0; i<size; i++)
    Push(lst,tab[i]);
  for(int i=0; i<size; i++)
    tab[i]=Pop(lst);
}

int main() {
  int t[]= {1,2,3,4,5};
  inverseTable(t, 5);
  for(int i=0; i<5; i++)
    printf("%d\t",t[i]);
  return 0;
}
```

Exercice 10 : (TD/TP)

Écrivez la fonction **duplicateList** qui duplique les éléments d'une liste à l'aide d'une **file**

```
void duplicateList(Queue *Q) {
  int x,size=Q->size;
  for(int i=0; i<size; i++) {
    x=deQueue(Q);
    enQueue(Q,x);
    enQueue(Q,x);
  }
}

int main() {
  Queue *Q=initQ();
  for(int i=0; i<5; i++)
    enQueue(Q, i);
  duplicateList(Q);
  display_list(Q->first);
  return 0;
}
```