Ejercicios Listas

Ejercicio 2. Impresión en orden inverso. Implementar (con control de errores) las siguientes funciones para imprimir los elementos de una lista en orden inverso:

a) Función no recursiva que no use ningún TAD adicional:

```
int _list_printInverseSeq(FILE *fp, List *pl, p_list_ele_print print_elem){
  void *elem = NULL;
  int i, size, count;

  if (!fp || !pl) return -1;

  size = list_size(pl);
  for (i=0, count=0; i<size; i++){
    elem = list_popBack(pl);
    if (!elem) return -1;
    count += print_elem(fp, elem);
    list_pushFront(pl, elem);
  }

  return count;
}
```

b) Función no recursiva que use una pila:

```
int _list_printInverseStack(FILE *fp, List *pl, p_list_ele_print print_elem){
  Stack *s = NULL;
  void *elem = NULL;
  int i, size, count;

  if (!fp || !pl) return -1;

  if (!(s = stack_init())) return -1;

  while (list_isEmpty(pl) == FALSE)
    stack_push(s, list_popFront(pl));

  while (stack_isEmpty(s) == FALSE){
    elem = stack_pop(s);
    if (!elem) return -1;
    count += print_elem(fp, elem);
    list_pushFront(pl, elem);
  }

  stack_free(s);
  return count;
}
```

c) Función recursiva:

```
int _list_printInverseRec(FILE *fp, List *pl, p_list_ele_print print_elem){

  if (!fp || !pl) return -1;

  return _list_printInverseRecInner(fp, pl->first, print_elem);
}

int _list_printInverseRecInner(FILE *fp, Node *pn, p_list_ele_print print_elem){

  if (!fp) return -1;
```

```
  if (!pn) return 0;

  return _list_printInverseRecInner(fp, pn->next) + print_elem(fp,pn->info);
}
```

Ejercicio 3. Inversión de lista. Implementar una función que invierta una lista, de manera que el último nodo se convierta en el primero, y así sucesivamente.

```
Status _list_invert(List *pl){
  List *pm = NULL;
  Status st = OK;

  if (!pl) return ERROR;

  if (!(pm = list_init())) return ERROR;

  while (list_isEmpty(pl) == FALSE || st == OK)
    st = list_pushFront(pm, list_popBack(pl));

  while (list_isEmpty(pm) == FALSE || st == OK)
    st = list_pushBack(pl, list_popBack(pm));

  list_free(pm);

  return st;
}
```

Ejercicio 4. Intercambio de primer y último nodo. Implementar (con control de errores) las siguientes funciones para intercambiar el primer y el último elemento de una lista:

a) Función con acceso a la estructura de datos:

```
Status _list_swapFLDS(List *pl){
  Node *pn = NULL;
  void *elem = NULL;

  if (!pl) return ERROR;
  if (list_isEmpty(pl) || !(pl->first->next)) return OK;

  pn = pl->first;
  while (pn){
    if (!(pn->next)){
      elem = pl->first->info;
      pl->first->info = pn->info;
      pn->info = elem;
    }
    pn = pn->next;
  }

  return OK;
}
```

b) Función derivada (sin acceso a la estructura de datos):

```
Status _list_swapFLPrimitives(List *pl){
  void *elem1 = NULL, *elem2 = NULL;

  if (!pl) return ERROR;
  if (list_isEmpty(pl)) return OK;

  elem1 = list_popFront(pl);
  if (!(elem2 = list_popBack(pl)){
```

```
    list_pushFront(pl, elem1);
    return OK;
  }

  if (!list_pushFront(pl, elem2) || !list_pushBack(pl, elem1))
    return ERROR;

  return OK;
}
```

Ejercicio 5. Concatenación de listas. Implementar una función primitiva (con acceso a la estructura de datos) que enlace dos listas:

```
Status _list_concatenate(List *pl1, List *pl2){
  Node *pn = NULL;

  if (!pl1 || !pl2) return ERROR;

  if (list_isEmpty(pl1)){
    pl1->first = pl2->first;
    pl2->first = NULL;
    return OK;
  }

  pn = pl1->first;
  while (pn->next)
    pn = pn->next;

  pn->next = pl2->first;
  pl2->first = NULL;

  return OK;
}
```

Ejercicio 6. Inserción en lista ordenada. Implementar (con control de errores) una función para insertar en una lista ordenada decreciente:

```
Status list_pushSorted(List *pl, const void *e, p_list_ele_compare comp_elem){
  Node *pn = NULL, *aux = NULL;

  if (!pl || !e) return ERROR;

  pn = node_new(e);
  if (!pn) return ERROR;
  aux = pl->first;
  while (aux->next || comp_elem(aux->info, e) < 0)
    aux = aux->next;

  pn->next = aux->next;
  aux->next = pn;

  return OK;
}
```

Ejercicio 8. Combinación de listas ordenadas. Implementar (con control de errores) una función para fusionar dos listas ordenadas decrecientes, devolviendo una lista ordenada con los elementos de ambas:

```
List *_list_merge(List *pl1, List *pl2, p_list_ele_compare comp_elem){
  List *new = NULL;
  Node *pn1 = NULL, *pn2 = NULL;

  if (!pl1 || !pl2) return NULL;
```

```
  new = list_init();
  if (!new) return NULL;
  pn1 = pl1->first;
  pn2 = pl2->first;

  if (!pn1 || !pn2) return new;

  while (pn1 && pn2){
    if (comp_elem(pn1->info, pn2->info) <= 0){
      list_pushBack(new, pn1->info);
      pn1 = pn1->next;
    }
    else if (comp_elem(pn1->info, pn2->info) > 0){
      list_pushBack(new, pn2->info);
      pn2 = pn2->next;
    }
  }

  if (!pn1){
    while (pn2){
      list_pushBack(new, pn2->info);
      pn2 = pn2->next;
    }
    return new;
  }
  if (!pn2){
    while (pn1){
      list_pushBack(new, pn1->info);
      pn1 = pn1->next;
    }
    return new;
  }

  return new;
}
```

Ejercicio 9. Lista con acceso posterior. Sea la siguiente estructura de datos para implementar una lista enlazada:

```
struct _RAList{
  Node *first; /*First node of the list.*/
  Node *last;  /*Last node of the list.*/
};
```

Implementar las siguientes primitivas:

```
RAList *ralist_new(){
  RAList *pl = NULL;

  pl = malloc(sizeof(RAList));
  if (!pl) return NULL;

  pl->first = NULL;
  pl->last = NULL;

  return pl;
}

Bool ralist_isEmpty(const RAList *pl){
  if (!pl) return TRUE;

  if (!pl->first || !pl->last) return TRUE;

  return FALSE;
}
```

```
Status ralist_pushFront(RAList *pl, const void *e){
  Node *pn = NULL;

  if (!pl || !e) return ERROR;

  pn = node_new();
  if (!pn) return ERROR;
  pn->info = (void*)e;

  if (ralist_isEmpty(pl)){
    pl->first = pn;
    pl->last = pn;
    return OK;
  }

  pn->next = pl->first;
  pl->first = pn;

  return OK;
}

Status ralist_pushBack(RAList *pl, const void *e){
  Node *pn = NULL;

  if (!pl || !e) return ERROR;

  pn = node_new();
  if (!pn) return ERROR;
  pn->info = (void*)e;

  if (ralist_isEmpty(pl)){
    pl->first = pn;
    pl->last = pn;
    return OK;
  }

  pl->last->next = pn;
  pl->last = pn;

  return OK;
}

void *ralist_popFront(RAList *pl){
  void *elem = NULL;
  Node *aux = NULL;

  if (!pl || ralist_isEmpty(pl)) return NULL;

  if (pl->first == pl->last){
    elem = pl->first->info;
    node_free(pl->first);
    return elem;
  }

  elem = pl->first->info;
  aux = pl->first;
  pl->first = pl->first->next;
  if (list_isEmpty(pl))
    pl->last = NULL;

  node_free(aux);

  return elem;
}

void *ralist_popBack(RAList *pl){
  void *elem = NULL;
  Node *pn = NULL;

  if (!pl || ralist_isEmpty(pl)) return NULL;
```

```
  if (pl->first == pl->last){
    elem = pl->first;
    node_free(pl->first);
    pl->first = NULL;
    pl->last = NULL;
    return elem;
  }

  elem = pl->last->info;
  pn = pl->first;
  while (pn->next->next)
    pn = pn->next;
  pl->last = pn;
  pn->next = NULL;
  node_free(pn);

  return elem;
}
```

Ejercicio 10. Lasta doblemente circular. Sea la siguiente estructura de datos para implementar una lista doblemente circular:

```
struct _Node{
  void *info;
  struct _Node *next;
  struct _Node *prev;
};
```

```
struct _DCList{
  Node *first;
};
```

a) Implementar la primitiva de inserción delante:

```
Status dclist_pushFront(DCList *pl, const void *e){
  Node *pn = NULL;

  if (!pl || !e) return ERROR;

  pn = node_new();
  if (!pn) return ERROR;
  pn->info = (void*)e;

  if (dclist_isEmpty(pl)){
    pl->first = pn;
    pn->next = pn;
    pn->prev = pn;
    return OK;
  }

  pn->next = pl->first;
  pn->prev = pl->first->prev;
  pn->prev->next = pn;
  pl->first->prev = pn;
  pl->first = pn;

  return OK;
}
```

b) Implementar la primitiva de inserción detrás:

```
Status dclist_pushBack(DCList *pl, const void *e){
  Node *pn = NULL;

  if (!pl || !e) return ERROR;

  pn = node_new();
  if (!pn) return ERROR;
  pn->info = (void*)e;

  if (dclist_isEmpty(pl)){
    pl->first = pn;
    pn->prev = pn;
    pn->next = pn;
    return OK;
  }

  pn->next = pl->first;
  pn->prev = pl->first->prev;
  pn->prev->next = pn;
  pl->first->prev = pn;

  return OK;
}
```

c) Implementar la primitiva de extracción delante:

```
void *dclist_popFront(DCList *pl){
  void *elem = NULL;
  Node *pn_prev = NULL, *pn_next = NULL;

  if (!pl || dclist_isEmpty(pl)) return NULL;

  if (pl->first->next == pl->first){
    elem = pl->first->info;
    node_free(pl-first);
    pl->first = NULL;
    return elem;
  }

  pn_prev = pl->first->prev;
  pn_next = pl->first->next;

  pn_prev->next = pn_next;
  pn_next->prev = pn_prev;
  elem = pl->first->info;
  free(pl->first);
  pl->first = pn_next;

  return elem;
}
```

d) Implementar la primitiva de extracción detrás:

```
void *dclist_popBack(DCList *pl){
  void *elem = NULL;
  Node *pn_prev = NULL, *pn_next = NULL;

  if (!pl || dclist_isEmpty(pl)) return NULL;

  if (pl->first == pl->first->next){
    elem = pl->first->info;
    node_free(pl->first);
    pl->first = NULL;
    return elem;
  }
```

```
  pn_prev = pl->first->prev->prev;
  pn_next = pl->first;

  pn_prev->next = pn_next;
  pn_next->prev = pn_prev;
  elem = pl->first->prev->info;
  node_free(pl->first->prev);

  return elem;
}
```