Comenzado el	sábado, 12 de septiembre de 2020, 09:32
Estado	Finalizado
Finalizado en	sábado, 12 de septiembre de 2020, 11:12
Tiempo empleado	1 hora 40 minutos
Calificación	**0,5** de 2,0 (**23**%)

Pregunta 1

Finalizado

Puntúa 0,1 sobre 0,7

Dada la clase simplemente enlazada **sll_t<T>**, desarrollar el método **RECURSIVO** que invierte el orden de los elementos de dicha clase sin usar ninguna estructura de datos auxiliar.

El método a desarrollar sería **privado** y tendría la siguiente cabecera:

```
template <class T>
sll_node_t<T>* sll_t<T>::inv_r_(sll_node_t<T>* nodo)
```

El método público que invocaría a este método privado desde el exterior sería el siguiente:

```
template <class T>
void sll_t<T>::inv_r()
{ if (!empty()) inv_r_(head_);
}
```

Por ejemplo, si tenemos la siguiente **sll_t<int> L**:

1 → 2 → 3 → 4 → 5

e invocamos a **L.inv_r()**, el resultado esperado al mostrar la lista **L** en pantalla sería:

5 → 4 → 3 → 2 → 1

```
template <class T>
sll_node_t<T>* sll_t<T>::inv_r_(sll_node_t<T>* nodo)
{
        if(nodo == NULL)                        return NULL;    //no existe lista
        if(nodo ->get_next() == NULL)   return nodo;    // la lista es de un solo nodo
        sll_t<sll_node_t<T>*> list = inv_r(nodo->get_next());   //llamada recursiva de la lista
        nodo->get_next()->get_next() = node;                    //creamos el nodo y enlazamos co
        nodo->get_next() = NULL;                                // el primer nodo de la anterior
        return list;                                    // resto de los nodos
}
```

Solución:

```
template <class T>
sll_node_t<T>*
sll_t<T>::inv_r_(sll_node_t<T>* nodo)
{
  sll_node_t<T>* sig = nodo->get_next();
  if (nodo == head_)
    head_->set_next(NULL);

  if (sig == NULL)
  {
    head_ = nodo;
    return nodo;
  }
  sig = inv_r_(sig);
  sig->set_next(nodo);
  return nodo;
}
```

Pregunta 2

Finalizado

Puntúa 0,3 sobre 0,3

Considérese la función de Ackermann:

$$A(m,n) = \begin{cases} n+1, & \text{si } m = 0; \\ A(m-1,1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m-1, A(m,n-1)), & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Implementar un **algoritmo recursivo** que calcule el valor de la función para cualquier par de términos, y cuya cabecera sea:

```
int Ack(int m, int n);
```

```
void
int Ack(int m, int n)
{
        //caso base
        if(m == 0)                      return n+1;
        // caso generales
        if(m > 0 && n == 0)             return Ack(m-1, 1);
        if(m > 0 && n> 0)
        {
                int u = Ack(m, n-1);
                return Ack(m-1, u);
        }
}
```

Solución:

```
int Ack(int m, int n)
{ if (m == 0)
    return n + 1;
  else {
    if (n == 0) return Ack(m - 1, 1);
    else        return Ack(m - 1, Ack(m, n - 1));
  }
}
```

Pregunta 3

Finalizado

Puntúa 0,0 sobre 0,3

Se dispone de una clase de gestión de matrices **matrix_t** que dispone de los siguientes métodos:

- **get_m()**, que devuelve el número de filas de la matriz.
- **get_n()**, que devuelve el número de columnas de la matriz.
- **resize(m, n)**, que destruye el contenido de la matriz invocante y reserva memoria para **m** filas y **n** columnas.
- **item(i, j)**, que permite acceder, tanto para lectura como para escritura, al dato en la fila **i** y la columna **j** de la matriz, para $0 \leq i < m$ y $0 \leq j < n$, siendo **m** el número de filas y **n** el número de columnas de la matriz.

Usando estos métodos, se pide definir un método público de la clase **matrix_t** que convierta la matriz invocante en la concatenación de otras dos matrices de tipo **matrix_t**. Su cabecera debe ser la siguiente:

```
template<class T>
void matrix_t<T>::concatenate(const matrix_t<T>& u,
                              const matrix_t<T>& v,
                              const bool by_rows);
```

donde el parámetro **by_rows** indica si la concatenación se hace por filas o por columnas. Por ejemplo:

```
A : 2x3          B: 2x3
a  b  c          g  h  i
d  e  f          j  k  l

C.concatenate(A, B, true);
(A y B deben tener igual número de filas)

C : 2x6
a  b  c  g  h  i
d  e  f  j  k  l

D.concatenate(A, B, false);
(A y B deben tener igual número de columnas)

D : 4x3
a  b  c
d  e  f
g  h  i
j  k  l
```

```
template<class T>
void matrix_t<T>::concatenate(const matrix_t<T>& u, const matrix_t<T>& v, const bool by_rows)
{
        assert(u.get_m() == v.get_m()); //comprobamos si tienen el mismo n° de filas
        assert(u.get_n() == v.get_n());         //comprobamos si tienen el mismo n° de columnas
        resize(get_m(), get_n()*get_m());       //destruimos el contenido de la matriz invocante
        for(int i = 1; i <= u.get_m(); i++)
                for(int j = 1; j <= get_n(); j++)
                        item(i, j) = u.item(i, j) | v.item(i,j);
}
```

```
template<class T>
void matrix_t<T>::concatenate(const matrix_t<T>& u,
                              const matrix_t<T>& v,
                              const bool by_rows)
{
  if (by_rows)  // por filas
  {
    assert(u.get_m() == v.get_m());  // Igual número de filas
    resize(u.get_m(), u.get_n() + v.get_n());

    for (int i = 0; i < get_m(); i++) {  // Para cada fila

      for (int j = 0; j < u.get_n(); j++)  // Copiar todas las columnas de u
        item(i, j) = u.item(i, j);

      for (int j = 0; j < v.get_n(); j++)  // seguidas de todas las de v
        item(i, j + u.get_n()) = v.item(i, j);
    }
  }
  else  // por columnas
  {
    assert(u.get_n() == v.get_n());  // Igual número de columnas
    resize(u.get_m() + v.get_m(), u.get_n());

    for (int i = 0; i < u.get_m(); i++)  // Copiar todas las filas de u
      for (int j = 0; j < get_n(); j++)
        item(i, j) = u.item(i, j);

    for (int i = 0; i < v.get_m(); i++)  // seguidas de todas las de v
      for (int j = 0; j < get_n(); j++)
        item(i + u.get_m(), j) = v.item(i, j);
  }
}
```

Pregunta 4

Finalizado

Puntúa 0,1 sobre 0,7

Sea un programa que gestiona vectores dispersos basados en listas enlazadas simples, mediante las siguientes clases.

- La clase **sparse_node** contiene los atributos públicos **inx** (int), **val** (double) y **next** (sparse_node*), tal que el elemento en posición **inx** del vector disperso tendría valor **val**, y el puntero **next** apuntaría al siguiente nodo de la lista enlazada correspondiente. La clase tiene dos constructores: **sparse_node**(int, double), que copia los valores de los argumentos a **inx** y **val**, y **sparse_node**(sparse_node*), que copia los valores del nodo dado por argumento.
- La clase **sparse_vector** que contiene los atributos **head** (sparse_node*), que apunta al primer elemento de la lista enlazada que compone el vector, y **size** (int), que indica el tamaño del vector denso correspondiente. La clase contiene el método **empty**() que vacía todo su contenido, e **insert_tail**(sparse_node*), que enlaza el nodo dado por argumento al final de la lista enlazada interna.

Se pide diseñar un método

```
void sparse_vector::add(const sparse_vector& u, const sparse_vector& v);
```

tal que el vector invocante adquiera el valor de la suma de los vectores **u** y **v**, que deben tener igual tamaño. El algoritmo es el siguiente:

- Si dos elementos tienen igual índice (inx) tanto en **u** como en **v**, se suman sus valores (val) y se introduce el resultado con el mismo índice en el vector resultado.
- Si no tienen igual índice, se copia el de menor índice al vector resultado y se avanza al siguiente elemento (next) del vector correspondiente.
- Cuando se llegue al final de **u** o de **v**, se deberá copiar el resto del otro vector al vector resultado.

Para simplificar el proceso, se asume que los elementos de los vectores dispersos siempre están ordenados de menor a mayor índice.

```
void sparse_vector::add(const sparse_vector& u, const sparse_vector& v)
{
        assert(u.size() == v.size());   //punto: deben tener igual tamaño
        resize(get_size());             //punto: el vector invocante debe tener el mismo tamaño
        sll_node_t<T> *aux = head_;
        while(aux != NULL)
        {
                if(u.get_val().get_inx() ==v.get_val().get_inx())
                {
                        for(int i= 0; i < get_size(); i++)
                        {
                                sparce_node(get_inx(), u.get_inx().get_val() + v.get_inx().get_v
                                aux = aux -> get_next(head_);
                        }
                }
                else{


        }
}
```

```
void sparse_vector::add(const sparse_vector& u, const sparse_vector& v)
{
  assert(u.size == v.size);
  empty();
  size = u.size;
  sparse_node *i = u.head, *j = v.head;
  while (i != NULL && j != NULL)
  {
    if (i->inx == j->inx) {
      insert_tail(new sparse_node(i->inx, i->val + j->val));
      i = i->next;
      j = j->next;
    } else if (i->inx < j->inx) {
      insert_tail(new sparse_node(i));
      i = i->next;
    } else {
      insert_tail(new sparse_node(j));
      j = j->next;
    }
  }
  if (i == NULL)
    while (j != NULL) {
      insert_tail(new sparse_node(j));
      j = j->next;
    }
  else
    while (i != NULL) {
      insert_tail(new sparse_node(i));
      i = i->next;
    }
}
```

◄ ¿Te vas a presentar al examen de teoría (prueba objetiva) de SEPTIEMBRE?

Ir a...

Examen práctico de reserva 16 de septiembre 2020 ►