Examen de prácticas (C3)

Programación-II - Ingeniería Robótica

2021.05.31

Información

- El examen tendrá una duración máxima de 2 horas, desde las 09:30 a las 11:30.
- Todos los ejercicios se escribirán en lenguaje **C++**.

  Se corregirán con la versión del compilador instalado en los laboratorios de la EPS.

  Puedes ayudarte de estas referencias sobre C++.
- El `Makefile` que tienes disponible te permite crear el archivo para la entrega así: `make tgz`, como has hecho en las prácticas. También te permite compilar cada una de las dos *preguntas* por separado así: `make p1` y `make p2` o una tras otra así: `make`.
- Dispones de un programa principal de prueba para cada pregunta: `mainp1.cc` y `mainp2.cc`.
- Un error de compilación/enlace implicará un cero en la pregunta donde se produzca, por tanto **asegúrate de que tu código compila correctamente aunque determinadas funciones no hagan nada o no lo hagan bien**.
- Puedes hacer tantas entregas como quieras, sólo se corregirá la última. Las entregas son similares a las que has hecho durante el curso con las prácticas: https://pracdlsi.dlsi.ua.es. Recuerda ir haciendo entregas parciales mientras esté abierto el plazo de entrega, no se admiten entregas por ningún otro cauce ni fuera de plazo.
- Para que te hagas una idea de la cantidad de código a escribir, cada archivo `.cc` pedido ocupa, más o menos, esta cantidad de líneas:

25	p1.cc
25	node.cc
100	list.cc

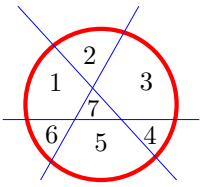
Si a ti te ocupan un número distinto de líneas, es normal, no pasa nada.

Ejercicio 1 (4 puntos)

Tus amigos y tú, en total `'n'` personas, habéis comprado una pizza y queréis saber cuál es el número mínimo de cortes que hay que hacerle para que cada uno de vosotros tenga al menos un trozo para comer -*no importa el tamaño del mismo*-.

Los cortes no tienen por que pasar por el centro y son en línea recta, por ejemplo:

- Con ningún corte podría comer una persona.
- Con un corte podrían comer hasta dos personas.
- Con dos cortes podrían comer hasta cuatro personas.
- Con tres cortes podrían comer hasta siete personas...compruébalo:



- Y así sucesivamente.

Se pide:
Crear en un archivo llamado `p1.cc` el código de una función con el siguiente prototipo:

```
uint32_t C3::ncuts(uint32_t n);
```

la cual **devolverá el número mínimo de cortes** a hacer en la pizza de manera que a cada una de las `n` personas, *valor que se le pasa como parámetro*, les corresponda, al menos, un trozo.
Recuerda:

- La función `ncuts` **no imprime nada** y pertenece al espacio de nombres `C3`.
- Una vez creado `p1.cc` puedes comprobar si compila bien así: `make p1.o`
- Dispones de un programa principal de prueba (`mainp1.cc`) y puedes compilarlo con ayuda del `Makefile` así: `make p1`, y ejecutarlo así: `make runp1` o bajo valgrind así: `make valgrindp1`, de este modo comprobarás que no deja memoria sin liberar.

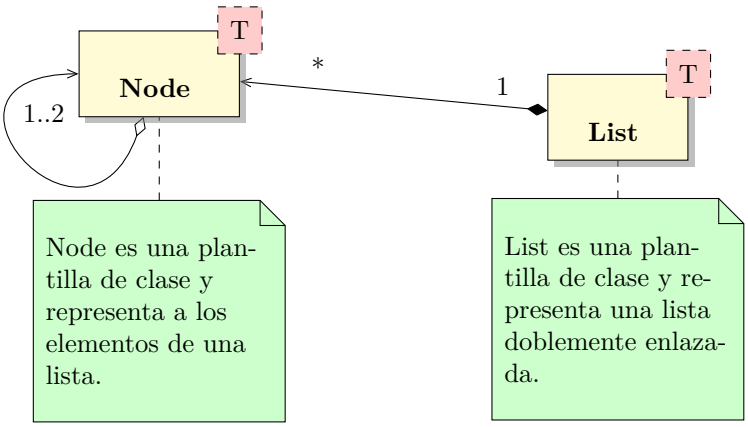
  La salida que produce es esta:

```
For 0 people you need 0 cuts!
For 1 people you need 0 cuts!
For 2 people you need 1 cuts!
For 3 people you need 2 cuts!
For 4 people you need 2 cuts!
For 5 people you need 3 cuts!
For 6 people you need 3 cuts!
For 7 people you need 3 cuts!
For 8 people you need 4 cuts!
For 9 people you need 4 cuts!
```

- Puedes crear las funciones auxiliares que necesites en `p1.cc`.

Ejercicio 2 (6 puntos)

1. Dada la siguiente estructura de clases empleada en un TAD *Lista doblemente enlazada* y representada mediante este diagrama **UML**:



2. Y siendo el contenido de los ficheros de cabecera el siguiente:

Contenido de `node.h`:

```
1  // -*- mode: c++ -*-
2
3  #ifndef NODE_H
4  #define NODE_H
5
6  namespace C3 {
7    // Declaracion adelantada
8    template<typename T> class Node;
9    // typedef con genericidad
10   template<typename T> using NodePtr = Node<T>*;
11
12   template<typename T>
13   class Node {
14   public:
15     Node(const T& k);
16     ~Node();
17     T& get_key();
18     const T& get_key() const;
19     NodePtr<T> get_next() const;
20     void set_next(NodePtr<T> n);
21     NodePtr<T> get_prev() const;
22     void set_prev(NodePtr<T> n);
23     bool has_next() const;
24     bool has_prev() const;
25   private:
26     T key;
27     NodePtr<T> next;
28     NodePtr<T> prev;
29   };
30 }  // C3
31
32 #endif /* NODE_H */
```

Contenido de `list.h`:

```
// -*- mode: c++ -*-

#ifndef LIST_H
#define LIST_H

#include <cstdint>
#include "node.h"

namespace C3 {
  // typedef con genericidad
  template<typename T> using fn_t = void (*)(NodePtr<T>);
  // Declaracion adelantada
  template<typename T> class List;

  template<typename T>
  class List {
  public:
    List();
    ~List();
    uint32_t length() const;
    bool insert_at(uint32_t p, const T& k);
    bool insert_front(const T& k);
    bool insert_back(const T& k);
    void for_each(fn_t<T> f);
    void for_each_reverse(fn_t<T> f);
  private:
    NodePtr<T> first;
    NodePtr<T> last;
    uint32_t nNodes;
  };
}  // C3

#endif /* LIST_H */
```

Se pide:

- Escribir el contenido de los archivos `node.cc` y `list.cc` como en las prácticas hechas durante el curso.
- Para ello debes implementar todos los métodos indicados en el fichero

`.h` correspondiente.

Aclaraciones:

1. Ambas clases pertenecen al *espacio de nombres* `C3`.
2. Al igual que ocurría con las *listas simplemente enlazadas*, en las *doblemente enlazadas* las posiciones válidas van de la `1` en adelante.
3. Los métodos de la clase `Node<T>` hacen lo siguiente:
   - `Node<T>::Node(const T& k)` : Inicializa un nodo a partir de un dato `k` de tipo `T`. Repasa tu práctica 2 para ver cómo lo hacías y adáptalo al caso de una lista doblemente enlazada.
   - `Node<T>::~Node()` : No hace nada, pero debe estar definido.
   - `bool Node<T>::has_next()/has_prev() const` : Nos dicen si el nodo actual tiene sucesor/antecesor respectivamente.
   - `T& Node<T>::get_key()` : Devuelve la clave del nodo.
   - `const T& Node<T>::get_key() const` : Devuelve la clave del nodo sin posibilidad de modificarla.
   - `NodePtr<T>Node<T>::get_next()/get_prev() const` : Devuelven el puntero al nodo siguiente o anterior respectivamente.
   - `void Node<T>::set_next(NodePtr<T>)/set_prev(NodePtr<T>) const` : Modifican el puntero al nodo siguiente o anterior respectivamente.
4. Los métodos de la clase `List<T>` hacen lo siguiente:
   - `List<T>::List()` : Inicializa la *lista* a valores apropiados nada más ser creada.
   - `List<T>::~List()` : Libera toda la memoria de los nodos de la *lista*.
   - `uint32_t length() const` : Devuelve el número de nodos en la *lista*.
   - `bool insert_at(uint32_t p, const T& k)` : Inserta un dato `k` de tipo `T` en la posición `p` de la lista. Devuelve *cierto* si lo ha podido hacer, *falso* en otro caso. Recuerda que trabajamos con una *lista doblemente enlazada*.

- `bool insert_front(const T& k)` : Inserta un dato `k` de tipo `T` al frente de la lista. Devuelve *cierto* si lo ha podido hacer, *falso* en otro caso.
- `bool insert_back(const T& k)` : Inserta un dato `k` de tipo `T` al final de la lista. Devuelve *cierto* si lo ha podido hacer, *falso* en otro caso.
- `void for_each(fn_t<T> f)` : Aplica la función `f` a cada uno de los nodos de la lista, desde el primero hasta el último. Para ello llama a la función `f` pasándole como argumento el nodo correspondiente, uno tras otro. Pista: *piensa en un bucle for*.
- `void for_each_reverse(fn_t<T> f)` : Aplica la función `f` a cada uno de los nodos de la lista, desde el último hasta el primero. Para ello llama a la función `f` pasándole como argumento el nodo correspondiente, uno tras otro.

Ayuda:

- Implementa los nuevos archivos `.cc` en este orden, te será más sencillo:
  1. `node.cc`
  2. `list.cc`
- Crea inicialmente una versión *vacía* de cada función/método que devuelva un valor cualquiera del tipo de dato pedido, esto te permitirá tener código que compila aunque no haga todavía lo que se pide.
- Una vez creado un fichero `.cc` puedes comprobar si compila bien así:

```
make node.o
make list.o
```

- Puedes crear las funciones auxiliares que necesites tanto en `node.cc` como en `list.cc`.
- Dispones de un programa principal de prueba (`mainp2.cc`) y puedes compilarlo con ayuda del `Makefile` así: `make p2`, y ejecutarlo así: `make runp2` o `make valgrindp2`, de esta última forma comprobarás que no deja memoria sin liberar. La salida que produce es esta:

```
Forward:
[ a b c d e f g h i j * k l m n o p q r s t u v w x y ]
Backward:
] y x w v u t s r q p o n m l k * j i h g f e d c b a [
```

- Recuerda que para poder instanciar una plantilla el compilador debe tener acceso a su código fuente, por eso en `mainp2.cc` tienes un `#include "list.cc"`.

Requisitos técnicos

- Requisitos que tiene que cumplir este trabajo práctico para considerarse válido y ser evaluado (si no se cumple alguno de los requisitos la calificación será **cero**):

- Al principio de todos los ficheros fuente (`.cc`) entregados se debe incluir un comentario con el nombre y el NIF (o equivalente) de la persona que entrega el examen, como en el siguiente ejemplo:

  ```
  // NIF: 12345678X
  // NOMBRE: PEREZ GARCIA, ALEJANDRO
  ```

- El archivo entregado se llama `irp2-c3.tgz` (todo en minúsculas). En el estarán todos los ficheros `.cc` pedidos en una carpeta llamada `irp2-c3`.

  Si por algún motivo entregas los ficheros `.h`, `mainp1/2.cc`, `Makefile`, etc… no pasa nada.

- Las clases, métodos y funciones implementados se llaman como se indica en el enunciado (respetando en todo caso el uso de mayúsculas y minúsculas). También es imprescindible respetar estrictamente los textos y los formatos de salida que se indican en este enunciado.

- **Lugar y plazo de entrega :**

  La entrega se realiza en https://pracdlsi.dlsi.ua.es. Allí puedes ver el plazo de entrega.

  Puedes entregar el examen tantas veces como quieras mientras el plazo de entrega esté abierto, sólo se corregirá la última entrega (las anteriores no se borran).

  El usuario y contraseña para entregar prácticas es el mismo que se utiliza en UACloud.

Detección de plagios/copias

- El examen debe ser un trabajo original de la persona que lo entrega.
- En caso de detectarse indicios de copia en el examen entregado, se tomarán las medidas disciplinarias correspondientes, informando a la dirección del DLSI por si hubiera lugar a otras medidas disciplinarias adicionales.