Labo 4 – Liste chainée simple

Durée: 3 semaines

```
template < typename T > class LinkedList {
public:
  using value_type = T;
  using reference = T&;
  using const_reference = const T&;
  using pointer = T*;
  using const_pointer = const T*;
```

- Classe C++ générique
- Alias habituels de la Standard Template Library

```
private:
struct Node {
  value_type data;
  Node* next;
};
```

- Les nœuds de la chaine utilisent une structure Node imposée
- Une struct est une class dont les membres sont public par défaut
- Données stockées dans l'attribut data de type value_type
- Pointeur vers le nœud suivant stocké dans l'attribut next

```
struct Node {
  value_type data; Node* next;
  Node(const_reference data, Node* next = nullptr)
    : data(data), next(next)
    { cout << "(C" << data << ") "; }
  ~Node() { cout << "(D" << data << ") "; }
  Node(Node&)  = delete;
  Node(Node&&) = delete;
};
```

- Constructeurs et destructeur imposés pour permettre de traquer ce que vous faites
- Pas de copie ni de déplacement de nœuds
- Toute création de nœud doit avoir la forme

```
Node* n = new Node(val,ptr);
```

```
private:
  Node* head;
  size_t nbElements;
```

- Les seuls attributs de la classe LinkedList sont
  - Un pointeur head vers le nœud de tête
  - Un compteur nbElements qui stocke le nombre d'éléments
- Vous êtes responsables d'initialiser et de maintenir le contenu de ces attributs

```
LinkedList() : /* a completer */
  { /* a completer */ }
~LinkedList() { /* a completer */ }
```

- Le constructeur initialise l'objet
- Le destructeur libère la mémoire alloué dynamiquement par l'objet

```
size_t size() const noexcept { /* a completer */ }
```

- Renvoie le nombre d'éléments stockés dans l'objet

```
void push_front( const_reference value) { /* … */ }
reference front() { /* … */ }
const_reference front() const { /* … */ }
void pop_front( ) { /* … */ }
```

- Insertion, accès et suppression en tête.
- L'élément en tête est celui pointé par l'attribut `head`.
- front() doît être codé deux fois.
  - Si l'objet est const, il renvoie une const_reference.
  - Sinon, il renvoie une reference normale vers la donnée stockée en tête, ce qui permet d'en changer le contenu.
- front() et pop_front() doivent lever une exception de type std::runtime_error si la liste est vide
- push_front() transmettent éventuellement une exception levée par le constructeur de copie de value_type, ou un bad_alloc lors de l'allocation dynamique du nouveau nœud

```
void insert( const_reference value, size_t pos ) {
  /* … */  }
reference at(size_t pos) { /* … */ }
const_reference at(size_t pos) const { /* … */ }
void erase( size_t pos ) { /* … */}
```

- Insertion, accès et suppression en position pos.
- pos == 0 signifie l'élément en tête.
- `pos == size()` est valide pour l'insertion en queue
- Toutes ces méthodes doivent lever une exception de type std::out_of_range si `pos` ne correspond pas à une position valide.
- insert() transmettent éventuellement une exception levée par le constructeur de copie de value_type ou un bad_alloc lors de l'allocation dynamique du nouveau nœud.

```
LinkedList( LinkedList& other ) { /* … */  }
LinkedList& operator= ( const LinkedList& other )
  { /* … */ }
```

- Constructeur de copie et opérateur d'affection
- Les deux doivent copier le contenu de la liste `other`
- L'opérateur d'affectation doit en plus
  - Vérifier qu'il ne s'agit pas d'une auto-affectation
  - Libérer la mémoire actuellement allouée par l'objet
- Ces opérations transmettent éventuellement une exception levée par le constructeur de copie de value_type ou un bad_alloc lors de l'allocation dynamique d'un nœud

```
size_t find( const_reference value ) const noexcept
{ /* a completer */ }
```

- Renvoie la position du premier élément de valeur égale à `value` en parcourant la liste depuis sa tête.
- Renvoie -1 si aucun élément de cette valeur n'est trouvé

```
void sort() { /* a completer */ }
```

- Trie les éléments de la liste par ordre croissant.
- Utiliser le tri par fusion
- Ne doit pas modifier les données mais uniquement les pointeurs entre les nœuds de la chaine.

```
template <typename T> std::ostream& operator <<
(std::ostream& os, const LinkedList<T>& liste) {
  os << liste.size() << ": ";
  auto n = liste.head;
  while (n) {
    os << n->data << " ";
    n = n->next;
  }
  return os;
}
```

- La fonction d’affichage du contenu de la liste vous est fournie

Tri par fusion

- Le tri par fusion doit se mettre en œuvre avec une fonction récursive traitant des parties de la liste
- Nous recommandons le prototype suivant pour cette fonction

```
Node*& mergeSort(Node*& start, size_t n);
```

- En entrée, `start` est une référence vers le pointeur du premier élément de la partie de la liste à traiter. `n` est le nombre d'éléments de cette partie de liste
- En sortie, `start` est éventuellement modifié si le premier élément n'est plus le même après tri. La fonction retourne également une référence vers le pointeur `next` du dernier élément de cette partie de liste. Il pointe typiquement vers le premier élément de la partie suivante à traiter.

Exception safety

- Toutes les méthodes pouvant lever ou laisser passer une exception doivent offrir une *garantie forte,* sans que cela n'aie d'impact sur la complexité de l'opération

- En cas d'échec,
  - L'objet doit conserver l'état qu'il avait avant l'appel à la méthode ou à l'opérateur
  - La mémoire éventuellement allouée pour rien doit être libérée.

- En cas d'exception qui n'est pas lancée par vous, vous devez la transmettre,
  - soit en la laissant passer
  - soit en la capturant puis en la relançant

Documentation

- Pour la soumission finale, toutes les méthodes, publiques ou privées, doivent être documentées dans le style doxygen

- Toutes les méthodes, publiques ou privées, doivent inclure un champ @remark qui indique la complexité de la méthode en fonction de ses paramètres et/ou des attributs de l’objet