```
/*
ESERCIZIO.

Definire un template di classe dList<T> i cui oggetti rappresentano una struttura dati lista doppiamente
concatenata (doubly linked list) per elementi di uno stesso tipo T. Il template dList<T> deve soddisfare i
seguenti vincoli:

1. Gestione della memoria senza condivisione.

2. dList<T> rende disponibile un costruttore dList(unsigned int k, const T& t) che costruisce una
lista contenente k nodi ed ognuno di questi nodi memorizza una copia di t.

3. dList<T> permette l'inserimento in testa ed in coda ad una lista in tempo O(1) (cioe` costante):
-- Deve essere disponibile un metodo void insertFront(const T&) con il seguente comportamento:
dl.insertFront(t) inserisce l'elemento t in testa a dl in tempo O(1).
-- Deve essere disponibile un metodo void insertBack(const T&) con il seguente comportamento:
dl.insertBack(t) inserisce l'elemento t in coda a dl in tempo O(1).

4. dList<T> rende disponibile un opportuno overloading di operator< che implementa l'ordinamento
lessicografico (ad esempio, si ricorda che per l'ordinamento lessicografico tra stringhe abbiamo che
"campana" < "cavolo" e che "buono" < "ottimo").

5. dList<T> rende disponibile un tipo iteratore costante dList<T>::const_iterator i cui oggetti permettono
di iterare sugli elementi di una lista.
*/

template<class T>
class dList {
private:

  class nodo {
  public:
    T info;
    nodo *prev, *next;
    nodo(const T& t, nodo* p = 0, nodo* n=0): info(t), prev(p), next(n) {}
  };

  nodo *first, *last; // puntatori al primo e ultimo nodo della lista
  // lista vuota IFF first == nullptr == last

  static void destroy(nodo* n) {
    if (n != nullptr) {
      destroy(n->next);
      delete n;
    }
  }

  static void deep_copy(nodo *src, nodo*& fst, nodo*& last) {
    if (src) {
      fst = last = new nodo(src->info);
      nodo* src_sc = src->next;
        while (src_sc) {
          last = new nodo(src_sc->info, last);
          last->prev->next = last;
          src_sc = src_sc->next;
        }
    }
    else {
      fst = last = nullptr;
    }
  }

  static bool isLess(const nodo* l1, const nodo* l2) {
    if(!l1 && !l2) return false;
    // l1 | l2
    if(!l1) return true; // vuota < non vuota
    if(!l2) return false; // non vuota < vuota
    // l1 & l2
    if(l1->info < l2->info) return true;
    else if(l1->info > l2->info) return false;
    else // l1->info == l2->info
      return isLess(l1->next, l2->next);
  }
```

```
public:

  dList(const dList& l) {
    deep_copy(l.first,first,last);
  }

  dList& operator=(const dList& l) {
    if(this != &l) {
      destroy(first);
      deep_copy(l.first,first,last);
    }
    return *this;
  }

  ~dList() { destroy(first); }

  void insertFront(const T& t) {
    first = new nodo(t,nullptr,first);
    if(first->next==nullptr) { // lista di invocazione era vuota
      last=first;
    }
    else { // lista di invocazione NON era vuota
      (first->next)->prev=first;
    }
  }

  void insertBack(const T& t) {
    if(last){ // lista non vuota
      last = new nodo(t,last,nullptr);
      (last->prev)->next=last;
    }
    else // lista vuota
      first=last=new nodo(t);
  }

  dList(unsigned int k, const T& t): first(nullptr), last(nullptr) {
    for(unsigned int j=0; j<k; ++j) insertFront(t);
  }

  bool operator<(const dList& l) const {
    if(this == &l) return false; // optimization: l < l e' sempre false
    return isLess(first, l.first);
  }

  class const_iterator {
    friend class dList <T>;
  private: // const_iterator indefinito IFF ptr==nullptr & past_the_end==false
    const nodo* ptr;
    bool past_the_end;

    // convertitore "privato" nodo* => const_iterator
    const_iterator(nodo* p, bool pte = false): ptr(p), past_the_end(pte) {}
  public:

    const_iterator(): ptr(nullptr), past_the_end(false) {}

    const_iterator& operator++() {
      if(ptr!= nullptr) {
        if(!past_the_end) {
          if(ptr->next != nullptr) {ptr = ptr->next;}
          else { ptr = ptr+1; past_the_end = true; }
        }
      }
      return *this;
    }

    const_iterator operator++(int){
      const_iterator aux(*this);
      if(ptr!= nullptr) {
        if(!past_the_end) {
          if(ptr->next != nullptr) ptr = ptr->next;
          else {ptr = ptr+1; past_the_end = true;}
        }
      }
```

```
      return aux;
    }

    const_iterator& operator--() {
      if(ptr != nullptr) {
        if(ptr->prev == nullptr) ptr=nullptr;
        else if(!past_the_end) ptr = ptr->prev;
        else {ptr = ptr-1; past_the_end = false;}
      }
      return *this;
    }

    const_iterator operator--(int){
      const_iterator aux(*this);
      if(ptr != nullptr) {
        if(ptr->prev == nullptr) ptr=nullptr;
        else if(!past_the_end) ptr = ptr->prev;
        else {ptr = ptr-1; past_the_end = false;}
      }
      return aux;
    }

    bool operator==(const const_iterator& cit) const {
      return ptr == cit.ptr;
    }

    bool operator!=(const const_iterator& cit) const {
      return ptr != cit.ptr;
    }

    const T& operator*() const {
      return ptr->info;
    }

    const T* operator->() const {
      return &(ptr->info);
    }
  };

  const_iterator begin() const {
    return const_iterator(first);
  }

  const_iterator end() const {
    if(!last) return const_iterator();
    return const_iterator(last+1,true); // attenzione: NON e' past the end
  }

};

// esempio d'uso
#include<iostream>

int main() {
  dList<int> x(4,2), y(0,0), z(6,8);
  y=x;
  x.insertFront(-2); z.insertFront(3); y.insertFront(0);
  if(x<y) std::cout << "x < y" << std::endl;
  if(z<x) std::cout << "z < x" << std::endl;
  if(y<z) std::cout << "y < z" << std::endl;
  if(z<y) std::cout << "z < y" << std::endl;

  std::cout << "x= ";
  dList<int>::const_iterator j = --(x.end());
  for(; j != x.begin(); --j) std::cout << *j << ' ';
  std::cout << *j << std::endl << "y= ";
  for(dList<int>::const_iterator k = y.begin(); k != y.end(); ++k) std::cout << *k << ' ';
  std::cout << std::endl << "z= ";
  dList<int>::const_iterator i = z.begin();
  for( ; i != z.end(); ++i) std::cout << *i << ' ';
  std::cout << std::endl;
}
```