LaB 3: Stack and Queue

- Stack

```
namespace stack{

    const unsigned int BLOCKDIM = 10;

    typedef int Elem;

    typedef struct {
        Elem* data;
        unsigned int size;
        unsigned int maxsize;
    }
      Stack;
```

createEmpty

```
Stack stack::createEmpty(){
   Stack sret;                          // creo lo stack . . .
   sret.size = 0;                       // . . . con size 0 (perché vuoto) . . .
   sret.maxsize = BLOCKDIM;             // . . . e maxsize BLOCKDIM (dichiarata nello struct) . . .
   sret.data = new Elem[sret.maxsize];  // alloco array di elem con dimensione maxsize (dove
                                           inserisco gli elementi)

   return sret;
}
```

push

// aggiunge elem in cima

```
void stack::push(const Elem el, Stack& st){
   if (st.size == st.maxsize) {                  // se lo stack è pieno . . .
      unsigned int newsize = st.maxsize + 10;    // . . . aggiungo 10 unità di spazio al maxsize
      Elem *temp = new Elem[newsize];            // creo un array temporaneo dove salverò gli
                                                    elementi già presenti nello stack

      // copio gli elementi dello stack su temp
      for (unsigned int i = 0; i < st.size; ++i) {
         temp[i] = st.data[i];
      }
      delete[] st.data;                          // cancello l'array vecchio contenente i dati con dimensione
                                                    più piccola

      st.data = temp;                            // l'array temporaneo diventa quello principale (con i dati)
      st.maxsize = newsize;                      // aumento maxsize
   }
   ++st.size;                                 // aumento size
   st.data[st.size - 1] = el;                 // l'ultimo elemento diventa l'elemento da inserire
}
```

pop

```
// toglie dallo stack l'ultimo elemento e lo restituisce

Elem stack::pop(Stack& st){
   if (st.size == 0) {                  // se lo stack è vuoto solleva una eccezione di tipo string
      string err = "errore";
      throw err;
   }
   Elem ret;
   ret = st.data[st.size - 1];      // salvo l'ultimo elemento
   --st.size;                       // diminuisco size (quindi elimino l'ultimo elemento)
   return ret;
}
```

top

```
// restituisce l'ultimo elemento dello stack senza toglierlo

Elem stack::top(Stack& st){
   if (st.size == 0) {                  // se lo stack è vuoto solleva una eccezione di tipo string
      string err = "errore";
      throw err;
   }
   Elem ret;
   ret = st.data[st.size - 1];      // salvo l'ultimo elemento
   return ret;                      // lo ritorno
}
```

- queue

```
//implementazione di una doubly linked list
struct queue::cell{
   Elem el;
   cell *next;
   cell *prev;
};
```

createEmpty

```
Queue queue::createEmpty(){
   Queue qret;                  // creo una nuova lista . . .
   qret.li = nullptr;           // . . . punto il primo elemento a null . . .
   qret.end = nullptr;          // . . . e l'ultimo elemento a null
   return qret;
}
```

enqueue

// inserisce l'elemento partendo dalla testa della coda

```
void queue::enqueue(Elem e, Queue& q){
   cell *aux = new cell;        // creo una nuova cella
   aux -> el = e;
   aux -> prev = nullptr;
   aux -> next = nullptr;
   if(isEmpty(q)) {             // se la queue è vuota
      q.li = aux;               // li è aux
      q.end = aux;              // l'ultimo elemento è aux
      return;
   }
   aux -> next = q.li;          // il next di aux è li (ossia il vecchio primo elemento)
   q.li -> prev = aux;          // collego il prev di li ad aux
   q.li = aux;                  // aux diventa il nuovo li (ossia il nuovo primo elemento)
}
```

// inserisce l'elemento partendo dalla fine della coda

```
void queue::enqueue(Elem e, Queue& q){
   cell *aux = new cell;        // creo una nuova cella
   aux -> el = e;
   aux -> prev = nullptr;
   aux -> next = nullptr;
   if(isEmpty(q)) {             // se la queue è vuota
      q.li = aux;               // li è aux
      q.end = aux;              // l'ultimo elemento è aux
      return;
   }

   q.end -> next = aux;
   aux -> prev = q.end;
   q.end = aux;
}
```

first

// restituisce l'elemento in prima posizione (se esiste) senza cancellarlo

```
Elem queue::first(Queue& q){
   if (isEmpty(q)) {
      string err = "errore";
      throw err;
   }
   return q.end -> el;          // ritorno l'elemento di end
}
```

Dequeue

```
// Cancella l'elemento (se esiste) dalla testa e lo restituisce

Elem queue::dequeue(Queue& q){
   Elem ret;
   if (isEmpty(q)) {
      string err = "errore";
      throw err;
   }
   ret = q.li -> el;                    // salvo il primo elemento della queue (elemento di li)
   cell *temp = q.li -> next;           // creo una cella temporanea con valore li -> next
   delete q.li;                         // elimino li (ossia primo elemento lista)
   q.li = temp;                         // li diventa il suo next (temp)
   return ret;
}
// Cancella l'elemento (se esiste) dalla coda e lo restituisce

Elem queue::dequeue(Queue& q){
   Elem ret;
   if (isEmpty(q)) {
      string err = "errore";
      throw err;
   }
   if (q.li == q.end) {                 // se c'è solo un elemento
      ret = q.end -> el;                        // salvo l'ultimo (e unico) elemento
      delete q.li;                              // elimino l'elemento
      q.li = nullptr;                           // quindi li è null
      q.end = nullptr;                          // quindi end è null
      return ret;                               // ritorno l'unico elemento
   }
   ret = q.end -> el;                  // salvo l'ultimo elemento
   cell *temp = q.end -> prev;         // creo una cella con valore penultimo elemento
   temp -> next = nullptr;             // con next nullo
   delete q.end;                       // elimino ultimo elemento
   q.end = temp;                       // l'ultimo elemento è ora temp (penultimo elemento)
   return ret;                         // ritorno l'ultimo elemento
}
```