DALHOUSIE UNIVERSITY
DEPARTMENT OF ENGINEERING MATHEMATICS
ENGM3282

ASSIGNMENT # 6, Due date: Tuesday, October 23, 2018, 1:00 PM

1. We sometimes see a stack implemented with a **sentinel** node which does not hold actual data but is used to locate the top of the stack.

   Write the necessary methods to complete the program `stacksentinel.cpp`

   For marking purposes, push 1, 2, 3, and 4 onto the stack, write the stack to the file, pop twice and write to the file.

```
// File: stacksentinal.cpp
// This program implements a simple stack of integers using a linked list
// with a sentinel

#include <iostream>
#include <fstream>
using namespace std;

class node {

friend class stack;  // stack needs access to node's members

private:
    int data;   // this is the data in a stack node
    node *next; // pointer to the next stack node

public:
    node(int x); // data = x, next = NULL
};

class stack {
private:
    node sentinel; // sentinel for the stack

public:
    stack(void);                          // constructor
    void push(int x);
    int pop(void);
    bool empty(void) const;               // check for empty stack
    void write(ostream &out) const;       // write the stack to out
};
/* A stack looks like a chain of nodes

     +------+     +------+     +------+          +------+
     |  0   |     | data |     | data |          | data |
     +------+     +------+     +------+          +------+
     | next |---->| next |---->| next |----> ... | next |---->NULL
     +------+     +------+     +------+          +------+
     sentinel      top of                        bottom of
                   stack                         stack

sentinel is an object which acts as a marker for the top of the stack.
sentinel.next points to the top of the stack which will be NULL if the stack
is empty. The value stored in sentinel.data is not part of the stack so we
can put any value there.  The value at the top of the stack is
sentinel.next->data. */
```

```
int main(void)
{
    stack mystack;
    ofstream fout ("stacksentinelout.txt");
    char ch;
    int x;

    cout << "A dynamic stack of integers\n";
    fout << "A dynamic stack of integers\n";

    do {
        // print a little menu
        cout << "\n\np = push \n";
        cout << "o = pop\n";
        cout << "s = print to screen\n";
        cout << "f = print to file\n";
        cout << "q = quit\n\n";

        cin >> ch;

        if (ch == 'p') {
            cout <<"\ndata to push :";
            cin >> x;
            mystack.push(x);
        }
        else if(ch == 'o') {
            if(mystack.empty()) {
                cout << "Stack is empty\n";
            } else {
                cout << "\n\ndata popped : " << mystack.pop();
            }
        }
        else if(ch == 's') mystack.write(cout);
        else if(ch == 'f') mystack.write(fout);

    }while(ch != 'q');

    fout.close();
    return 0;
}

node::node(int x)
{
    data = x;
    next = NULL;
}

/*  the push function takes an existing stack

     +------+     +------+     +------+          +------+
     |  0   |     | data |     | data |          | data |
     +------+     +------+     +------+          +------+
     | next |---->| next |---->| next |----> ... | next |---->NULL
     +------+     +------+     +------+          +------+
     sentinel      top of                        bottom of
                   stack                         stack
```

```
ptr points to a new node object

         +------+
         |   x  |
ptr ---->+------+
         | next |
         +------+

then connect this new node into the list

     +------+     +------+     +------+          +------+
     |  0   |     |   x  |     | data |          | data |
     +------+     +------+     +------+          +------+
     | next |---->| next |---->| next |----> ... | next |---->NULL
     +------+     +------+     +------+          +------+
     sentinel      new top      old top            bottom
*/

/* The pop function takes an existing stack

     +------+     +------+     +------+          +------+
     |  0   |     | data |     | data |          | data |
     +------+     +------+     +------+          +------+
     | next |---->| next |---->| next |----> ... | next |---->NULL
     +------+     +------+     +------+          +------+
     sentinel      top of                        bottom of
                   stack                         stack

and picks off the first node object to return it's data

          +------+
          | 1st  |
 ptr ---->+------+
          | next |
          +------+

then reassigns the next pointer to the second node

     +------+     +------+          +------+
     |  0   |     | 2nd  |          | data |
     +------+     +------+          +------+
     | next |---->| next |----> ... | next |---->NULL
     +------+     +------+          +------+
     sentinel      new top            bottom

*/
```

2. The following program implements a store and retrieve structure which is a cross between a stack and a queue. The retrieve member function retrieves the last node stored (LIFO like a stack) provided the number of nodes stored is odd otherwise it retrieves the first node stored (FIFO like a queue).

   Write the member functions of the `class stacq`.

   Hint: if `x` is odd then `x % 2` will be 1.

   For marking purposes store 1, 2, 3, 4 then print to the file, retrieve and print, retrieve and print.

```
// File: stacq.cpp
// This program implements a cross between a stack and a queue

#include <iostream>
#include <fstream>
using namespace std;

class node {
friend class stacq;

private:
   int data;   // this is the data in a stacq element
   node *next; // pointer to the next node on the stack

public:
   node(int x) {data = x; next = NULL;}

};

class stacq {
private:
   int count;   // number of values stored
   node* top;   // pointer to the top of the stacq

public:
   stacq(void);                    // constructor of an empty stacq
   void store(int x);              // store the value x
   int retrieve(void);             // retrieve a value
   bool empty(void) const;         // check for empty stack
   void write(ostream &out) const; // write stacq to out
};

/* A stacq  looks like

count =
         +------+     +------+     +------+          +------+
         | data |     | data |     | data |          | data |
top ---> +------+     +------+     +------+          +------+
         | next |---->| next |---->| next |----> ... | next |---->NULL
         +------+     +------+     +------+          +------+
*/

int main(void)
{
   stacq mine;
   ofstream fout ("stacqout.txt");
   char ch;
   int x;

   cout << "A dynamic stacq of integers\n";
```

```
   fout << "A dynamic stacq of integers\n";

   do {
      // print a little menu
      cout << "\n\ns = store \n";
      cout << "r = retrieve\n";
      cout << "p = print to screen\n";
      cout << "f = print to file\n";
      cout << "q = quit\n\n";

      cin >> ch;

      if (ch == 's') {
         cout <<"\ndata to store :";
         cin >> x;
         mine.store(x);
      }
      else if(ch == 'r') {
         if(mine.empty())
            cout << "empty\n";
         else
            cout << "\n\ndata retrieved :" << mine.retrieve();
      }
      else if(ch == 'p') mine.write(cout);
      else if(ch == 'f') mine.write(fout);

   }while(ch != 'q');

   fout.close();
   return 0;
}
```

3. Both a stack and a queue can be implemented using a data structure called a "Double Ended Queue" also called a deque (pronounced deck). A deque allows fast storage and retrieval at both ends of the list.

   Write the necessary functions to complete the program `deque.cpp`. You can assume that the queue is not empty when writing the `get_front` and `get_back` methods.

   For marking purposes `put_front` 1, 2 and 3; print the `deque`; `put_back` 4, 5 and 6; print the `deque`; `get_front`; print the `deque`; `get_back`; print the `deque` .

```
/* File: deque.cpp
   This program implements a double ended queue of integers as a
   doubly linked list */

#include <iostream>
#include <fstream>
using namespace std;

class node {
friend class deque;

private:
    int data;   // this is the data in a list element
    node *next; // pointer to the next node in the list
    node *prev; // pointer to the previous node in the list

public:
    node(int x); // data = x, prev=next = NULL
};

class deque
{
   private:
      node* front;      // pointer to the front of the list
      node* back;       // pointer to the back of the list

   public:
      deque(void);             // constructor of an empty queue
      void put_front(int x);   // put x at the front of the list
      void put_back(int x);    // put x at the back of the list
      int get_front(void);     // get the node at the front of the list
      int get_back(void);      // get the node at the back of the list
      bool empty(void) const;        // check for empty deque
      void write(ostream &out) const; // write data stored to out
};

/* A deque  looks like

         +------+     +------+     +------+          +------+
         | data |     | data |     | data |          | data |
         +------+     +------+     +------+          +------+
         | next |---->| next |---->| next |----> ... | next |---->NULL
NULL<----| prev |<----| prev |<----| prev |... <---- | prev |
         +------+     +------+     +------+          +------+
            ^                                           ^
            |                                           |
            |                                           |
          back                                        front
*/
```

```
int main(void)
{
    deque mydeque;
    ofstream fout ("dequeout.txt");
    char ch;
    int x;

    cout << "A dynamic deque of integers\n";
    fout << "A dynamic deque of integers\n";

    do {
        // print a little menu
        cout << "\n\n1 = put front \n";
        cout << "2 = put back \n";
        cout << "3 = get front\n";
        cout << "4 = get back\n";
        cout << "s = print to screen\n";
        cout << "f = print to file\n";
        cout << "q = quit\n\n";

        cin >> ch;

        if (ch == '1') {
            cout <<"\ndata to put front:";
            cin >> x;
            mydeque.put_front(x);
        }
        else if (ch == '2') {
            cout <<"\ndata to put back:";
            cin >> x;
            mydeque.put_back(x);
        }
        else if(ch == '3') {
            if(mydeque.empty()) {
                cout << "deque is empty\n";
            } else {
                cout << "\n\ndata gotten : " << mydeque.get_front();
            }
        }
        else if(ch == '4') {
            if(mydeque.empty()) {
                cout << "deque is empty\n";
            } else {
                cout << "\n\ndata gotten : " << mydeque.get_back();
            }
        }
        else if(ch == 's') mydeque.write(cout);
        else if(ch == 'f') mydeque.write(fout);

    }while(ch != 'q');

    fout.close();
    return 0;
}

node::node(int x)
{
    data = x;
    next = NULL;
    prev = NULL;
```

```
}

/*  the put_back function takes an existing deque

            +------+     +------+          +------+
            | last |     | data |          | 1st  |
            +------+     +------+          +------+
            | next |---->| next |----> ... | next |---->NULL
NULL<----   | prev |<----| prev |<---- ... | prev |
            +------+     +------+          +------+
               ^                               ^
               |                               |
               |                               |
             back                            front

ptr points to a new node

         +------+
         |   x  |
ptr ---->+------+
         | next |
         | prev |
         +------+

then connect this  new node into the list

         +------+     +------+     +------+          +------+
         |  x   |     | last |     | data |          | 1st  |
         +------+     +------+     +------+          +------+
         | next |---->| next |---->| next |----> ... | next |---->NULL
NULL<----| prev |<----| prev |<----| prev |<---- ... | prev |
         +------+     +------+     +------+          +------+
            ^                                           ^
            |                                           |
            |                                           |
          back                                        front
*/

/*  the put_front function takes an existing deque

            +------+     +------+          +------+
            | last |     | data |          | 1st  |
            +------+     +------+          +------+
            | next |---->| next |----> ... | next |---->NULL
NULL<----   | prev |<----| prev |... < ----| prev |
            +------+     +------+          +------+
               ^                               ^
               |                               |
               |                               |
             back                            front

ptr points to a new node

         +------+
         |   x  |
ptr ---->+------+
         | next |
         | prev |
         +------+
```

```
then connect this  new node into the list
         +------+     +------+         +------+      +------+
         | last |     | data |         | 1st  |      |  x   |
         +------+     +------+         +------+      +------+
         | next |---->| next |---->... | next |----> | next |---->NULL
NULL<----| prev |<----| prev |... <----| prev |<---- | prev |
         +------+     +------+         +------+      +------+
            ^                                          ^
            |                                          |
            |                                          |
           back                                      front
*/

/* The get_front function takes an existing deque

         +------+     +------+         +------+      +------+
         | last |     | data |         | 2nd  |      | 1st  |
         +------+     +------+         +------+      +------+
         | next |---->| next |---->... | next |----> | next |---->NULL
NULL<----| prev |<----| prev |... <----| prev |<---- | prev |
         +------+     +------+         +------+      +------+
            ^                                          ^
            |                                          |
            |                                          |
           back                                      front

and picks off the first data node to be returned

          +------+
          |  1st |
 ptr ---->+------+
          | next |
          | prev |
          +------+

then reassigns the front pointer

            +------+     +------+          +------+
            | last |     | data |          | 2nd  |
            +------+     +------+          +------+
            | next |---->| next |----> ... | next |---->NULL
NULL<----   | prev |<----| prev |... <---- | prev |
            +------+     +------+          +------+
               ^                               ^
               |                               |
               |                               |
              back                            front
*/

/* The get_back function takes an existing deque

         +------+     +------+     +------+          +------+
         | last |     | 2nd  |     | data |          | 1st  |
         +------+     +------+     +------+          +------+
         | next |---->| next |---->| next |----> ... | next |---->NULL
NULL<----| prev |<----| prev |<----| prev |<---- ... | prev |
         +------+     +------+     +------+          +------+
            ^                                          ^
            |                                          |
            |                                          |
```

```
           back                                          front

and picks off the last data node to be returned

          +------+
          | last |
 ptr ---->+------+
          | next |
          | prev |
          +------+

then reassigns the back pointer

            +------+     +------+          +------+
            | 2nd  |     | data |          | 1st  |
            +------+     +------+          +------+
            | next |---->| next |----> ... | next |---->NULL
NULL<----   | prev |<----| prev |<---- ... | prev |
            +------+     +------+          +------+
               ^                               ^
               |                               |
               |                               |
              back                            front
*/
```