Example of a Deque and Priority Queue as a Template

Posted by Alejandro G. Carlstein Ramos Mejia on October 15, 2010 February 9, 2012 About Programming / Algorithms / C++

Example of Deque, regular priority queue, and priority queue as template.

NOTIFICATION: These examples are provided for educational purposes. Using this code is under your own responsibility and risk. The code is given 'as is'. I do not take responsibilities of how they are used.

Deque.cc:

```
#ifndef DEQUE_CC_
#define DEQUE_CC_

#include <iostream>

using namespace std;

template <typename T> class Deque;

template <typename T>
class Node {

 private:

  // Contains the data of the linked Deque node
  T data;

  // Contains a pointer to the previous item in the linked Deque
  Node<T> *previous;

  // Contains a pointer to the next item in the linked Deque
  Node<T> *next;

 public:

         friend class Deque<T>;

  //The default constructor doesn't initialize the data members
  Node(void);

  // Create a linked Deque node by setting its data to the input parameter
  // and the next node in the linked Deque to NULL (0)
  Node(const T& new_data);

  Node(const T& new_data,
    Node<T>* new_previous,
```

```
    Node<T>* new_next);

  T get_data(void);

  Node<T>* get_previous(void);

  Node<T>* get_next(void);

  void set_data(T new_data);

  void set_previous(Node<T>* new_previous);

  void set_next(Node<T>* new_next);

  //The default destructor doesn't initialize the data members
  ~Node(void);

};

template <typename T>
class Deque {
 private:

  int num_elements;

  // Contains a pointer to the first node in the
  // linked Deque (the head of the Deque)
  Node<T> *head;

  // Contains a pointer to the last node in the
  // linked Deque (the head of the Deque)
  Node<T> *tail;

 public:

  // **** Overload Operators ****

  T& operator[] (const int index);

  Deque<T>& operator= (const Deque<T>& new_deque);

  // **** Constructors ****

  // Create a Linked Deque by initializing it's head to NULL (0)
  Deque(void);

  Deque(const Deque<T>& new_deque);

  // Delete all the elements in the linked Deque when the Deque is deleted
  ~Deque(void);

  // **** Get Methods ****
```

```
  bool is_empty(void) const;

  int size(void) const;

  // **** Set Methods ****

  // **** Print Methods ****

  // **** Methods ****

  void push_front(const T& new_data);

  void push_back(const T& new_data);

  void pop_front(void);

  void pop_back(void);

  T front(void) const;

  T back(void) const;

  void remove_at(int index);

  T& at(const int index);

  //erase

};

//////////////////////////////////////////////////////
// NODE PUBLIC METHODS
//////////////////////////////////////////////////////

// **** Constructors ****

template <typename T>
Node<T>::Node(void):
     previous(NULL),
     next(NULL){
}

template <typename T>
Node<T>::Node(const T& new_data):
     data(new_data),
     previous(NULL),
     next(NULL){
}

template <typename T>
Node<T>::Node(const T& new_data,
     Node<T>* new_previous,
     Node<T>* new_next):
```

```
     data(new_data),
     previous(new_previous),
     next(new_next){

}

// **** Get Methods ****
template <typename T>
T Node<T>::get_data(void){
 return data;
}

template <typename T>
Node<T>* Node<T>::get_previous(void){
 return previous;
}

template <typename T>
Node<T>* Node<T>::get_next(void){
 return next;
}

// **** set Methods ****
template <typename T>
void Node<T>::set_data(T new_data){
 data = new_data;
}

template <typename T>
void Node<T>::set_previous(Node<T>* new_previous){
 previous = new_previous;
}

template <typename T>
void Node<T>::set_next(Node<T>* new_next){
 next = new_next;
}

// **** Print Methods ****

// **** Methods ****

// **** Destructor ****

template <typename T>
Node<T>::~Node(void) {
}

////////////////////////////////////////////////////////
// DEQUE OVERLOAD OPERATORS
////////////////////////////////////////////////////////
template <typename T>
T& Deque<T>: 😮 perator[] (const int index){
```

```
 int i  = 0;

 Node<T>* current_node;

 if (index > -1 && index < num_elements && !is_empty()){

  // If index is between the middle and the end of the list, begin  from the end of
the list
  // else
  // (index is between the middle and the begin of the list), begin from the beginning
of the list
  if (index >= (num_elements / 2) ){

   current_node = tail;

   for (i = num_elements - 1;
    i > index;
    i--, current_node = current_node->previous);

  }else{

   current_node = head;

   for (i = 0;
    i < index;
    i++,current_node = current_node->next);

  }// end if

  return current_node->data;

 }else{

  //cerr << '[X] ERROR: Index out of bounds : ' << index << '/' << (num_elements) <<
endl;

 }// end if

 return current_node->data;
}

/**
 * operator =
 * @description: deep copy deque
 * @param: new_deque
 * @return: Deque&
 */
template <typename T>
Deque<T>& Deque<T>:😮perator= (const Deque<T>& new_deque)
{

 num_elements = 0;
```

```
    if ( new_deque.head != NULL ){

  Node<T> *current_node = new_deque.head;

  while (current_node != NULL){

   push_back(current_node->data);

   current_node = current_node->next;

   num_elements++;
  }
 }

 // return the existing object
    return *this;
}

//////////////////////////////////////////////////////
// DEQUE PRIVATE METHODS
//////////////////////////////////////////////////////

//////////////////////////////////////////////////////
// DEQUE PUBLIC METHODS
//////////////////////////////////////////////////////

// **** Constructors ****
template <typename T>
Deque<T>: eque(void):
          num_elements(0),
    head(NULL),
    tail(NULL){
}

template <typename T>
Deque<T>: eque(const Deque<T>& new_deque){

 num_elements = 0;

    if ( new_deque.head != NULL ){

  Node<T> *current_node = new_deque.head;

  while (current_node != NULL){

   push_back(current_node->data);

   current_node = current_node->next;

   num_elements++;
  }
 }
```

```
}

// **** Get Methods ****

template <typename T>
bool Deque<T>::is_empty() const{
 return((num_elements < 1));
}

template <typename T>
int Deque<T>::size() const{
 return num_elements;
}
// **** set Methods ****

// **** Print Methods ****

// **** Methods ****

template <typename T>
void Deque<T>: ush_front(const T& new_data){

    Node<T> *new_node;

    if (head == NULL) {

   new_node = new Node<T>(new_data, NULL, NULL);

   head = new_node;

   tail = new_node;

    } else {

  Node<T> *next_node = head;

  new_node = new Node<T>(new_data, NULL, next_node);

  head = new_node;

  next_node->set_previous(new_node);

    }

    num_elements++;

}

// Takes an item as a parameter, appends that item to
// the back of the Deque, and returns the Deque
template <typename T>
void Deque<T>: ush_back(const T& new_data){
```

```
    Node<T> *new_node;

    if (tail == NULL){

  new_node = new Node<T>(new_data, NULL, NULL);

  tail = new_node;

  head = new_node;

    } else {

  Node<T> *previous_node = tail;

  new_node = new Node<T>(new_data, previous_node, NULL);

  tail = new_node;

  previous_node->set_next(new_node);

    }// end if

    num_elements++;

}

template <typename T>
void Deque<T>:😡op_front(void){

 if (!is_empty()){

  Node<T> *new_head = head->get_next();

  if (new_head != NULL){

   new_head->set_previous(NULL);

   delete head;

   head = new_head;

   num_elements--;

  }

 }//end if

}

template <typename T>
void Deque<T>:😡op_back(void){

 if (!is_empty()){
```

```
  Node<T> *new_tail = tail->get_previous();

  if (new_tail != NULL){

   new_tail->set_next(NULL);

   delete tail;

   tail = new_tail;

   num_elements--;

  } //end if

 }//end if

}

template <typename T>
T Deque<T>::front() const{
 return (head->data);
}

template <typename T>
T Deque<T>::back() const{
 return (tail->data);
}

template <typename T>
void Deque<T>::remove_at(int index){

 bool is_found = false;

    Node<T> *current_node;

 current_node = head;
 for (int i = 0; current_node != NULL && !is_found; i++){

  if (i == index){

   //Remove from beginning
   if (current_node->previous == NULL){

    head = current_node->next;

      // Remove from the end
   }else if(current_node->next == NULL){
    current_node->previous->next  = NULL;

   }else{ //Remove from middle

    // Fix previous node's next to skip over the removed node
```

```
    current_node->previous->next = current_node->next;

    // Fix next node's previous to skip over the removed node
    current_node->next->previous = current_node->previous;

   }//end if

   delete current_node;

   is_found = true;

  }//end if

  current_node = current_node->next;

 }//end for

/*
 Node<T> *new_node;

    Node<T> *current_node = head;

 bool is_node_found = false;

 for (int i = 0;
   i < size() &&
   current_node != NULL
   && !is_node_found;
   i++){

  if (i == index){

   is_node_found = true;

   if (current_node == head){

    pop_front();

   }else if (current_node == tail){

    pop_back();

   }else{

    // In order to remove the current node, we have to conect the next node with
    // the node previous to the curent node.
    new_node = current_node->next;

    new_node->set_previous(current_node->get_previous());

    delete(current_node);

    current_node = new_node;
```

```
   }//end if

   num_elements--;

  }//end if

  current_node = current_node->next;

 }//end if
*/
}

template <typename T>
T& Deque<T>::at(const int index){

 int i  = 0;

 Node<T>* current_node;

 if (index > -1 && index < num_elements && !is_empty()){

  // If index is between the middle and the end of the list, begin  from the end of
the list
  // else
  // (index is between the middle and the begin of the list), begin from the beginning
of the list
  if (index >= (num_elements / 2) ){

   current_node = tail;

   for (i = num_elements - 1;
    i > index;
    i--, current_node = current_node->previous);

  }else{

   current_node = head;

   for (i = 0;
    i < index;
    i++,current_node = current_node->next);

  }// end if

  return current_node->data;

 }else{

  //cerr << '[X] ERROR: Index out of bounds : ' << index << '/' << (num_elements) <<
endl;

 }// end if
```

```
 return current_node->data;
}

// **** Destructor ****
template <typename T>
Deque<T>::~Deque(void) {

 Node<T> *next_node;

    Node<T> *current_node = head;

    while (current_node != NULL) {

        next_node = current_node->next;

        delete(current_node);

        current_node = next_node;

  num_elements--;
    }

}

#endif
```

PriorityQueue.cpp:

```
#include 'PriorityQueue.h'
// First Nodes Created With Constructor

int PriorityQueue::NumOfNodes=1;

// Constructor

PriorityQueue:🤔riorityQueue(void){

    Current.Previous = NULL;

    cout << 'Enter First Element of Queue'
         << endl;

    cin >> Current.Data;

    Current.Next = NULL;

    head = &Current;

    ptr = head;

}
```

```
// Function Finding Maximum Priority Element
int PriorityQueue::Maximum(void){

    int Temp;

    ptr = head;

    Temp = ptr->Data;

    while(ptr->Next != NULL){

        if(ptr->Data > Temp){

            Temp = ptr->Data;

  }

        ptr = ptr->Next;
    }

    if(ptr->Next == NULL && ptr->Data > Temp){

  Temp = ptr->Data;

 }

    return(Temp);
}

// Function Finding Minimum Priority Element
int PriorityQueue::Minimum(void){

    int Temp;

    ptr = head;

    Temp = ptr->Data;

    while(ptr->Next != NULL){

        if(ptr->Data < Temp){

            Temp = ptr->Data;

  }

  ptr = ptr->Next;

 }

 if(ptr->Next == NULL && ptr->Data < Temp){

        Temp = ptr->Data;
```

```
 }

 return(Temp);
}

// Function inserting element in Priority Queue
void PriorityQueue::Insert(int DT){

    struct Node *newnode;

    newnode = new Node;

    newnode->Data = DT;

    while(ptr->Next != NULL){

  ptr = ptr->Next;

 }

 if(ptr->Next == NULL){

        newnode->Next = ptr->Next;

  ptr->Next = newnode;

 }

 NumOfNodes++;
}

// Function deleting element in Priority Queue
int PriorityQueue:😬elete(int DataDel){

    struct Node *mynode,*temp;

    ptr = head;

    if(NumOfNodes == 1){

        cout << 'Cannot Delete the only Node'
             << endl;

  return FALSE;
    }

    if(ptr->Data == DataDel){

  /***  Checking condition for deletion of first node  ***/
        temp = ptr;

        ptr = ptr->Next;
```

```
      ptr->Previous = NULL;

      //delete temp;

      head = ptr;

      NumOfNodes--;

      return(TRUE);

  }else{

      while(ptr->Next->Next != NULL){

          /***  Checking condition for deletion of   ***/
          /*** all nodes except first and last node  ***/
          if(ptr->Next->Data == DataDel){

              mynode = ptr;

  temp = ptr->Next;

              mynode->Next = mynode->Next->Next;

              mynode->Next->Previous = ptr;

              delete temp;

              NumOfNodes--;

              return(TRUE);

 }

          ptr = ptr->Next;

}

      if(ptr->Next->Next == NULL &&
         ptr->Next->Data == DataDel){

 /***  Checking condition for deletion of last node  ***/
          temp = ptr->Next;

          delete temp;

          ptr->Next = NULL;

          NumOfNodes--;

          return(TRUE);
```

```
  }

 }

    return(FALSE);
}

// Function Searching element in Priority Queue
int PriorityQueue::Search(int DataSearch){

    ptr = head;

    while(ptr->Next != NULL){

        if(ptr->Data == DataSearch){

            return ptr->Data;

  }//end if

  ptr = ptr->Next;
    }

 if(ptr->Next == NULL &&
           ptr->Data == DataSearch){

  return ptr->Data;
 }

    return(FALSE);

}

// Function Displaying elements of Priority Queue
void PriorityQueue: isplay(void){

    ptr = head;

    cout << 'Priority Queue is as Follows:-'
         << endl;

    while(ptr != NULL){

        cout << ptr->Data
             << endl;

        ptr = ptr->Next;

 }

}

// Destructor of Priority Queue
```

```
PriorityQueue::~PriorityQueue(void){

 /* Temporary variable */
    struct Node *temp;

 while(head->Next != NULL){

        temp = head->Next;

  //    delete head;

        head = temp;
    }

    if(head->Next == NULL){
        delete head;
 }

}
```

PriorityQueue.h:

```
#ifndef PRIORITY_QUEUE_H_

#define PRIORITY_QUEUE_H_

#include <iostream>

using namespace std;

#include <iostream>

#include <cstdlib>

enum{

 FALSE = 0,

 TRUE = -1

};

////////////////////////////////

///// Implements Priority Queue

////////////////////////////////

// Class Prioriry Queue

class PriorityQueue{

 private:
```

```
  // Node of Priority Queue
  struct Node{
        struct Node *Previous;
        int Data;
        struct Node *Next;
  }Current;
  struct Node *head;  // Pointer to Head
  struct Node *ptr;
        // Pointer for travelling through Queue
  static int NumOfNodes;
        // Keeps track of Number of nodes
 public:
  PriorityQueue(void);
  int Maximum(void);
  int Minimum(void);
  void Insert(int);
  int Delete(int);
  void Display(void);
  int Search (int);
  ~PriorityQueue(void);
};
/*
//Main Function
void main()
{
    PriorityQueue PQ;
```

```
int choice;

int DT;

while(1)

{

    cout << 'Enter your choice'
         << endl;

    cout << '1. Insert an element'
         << endl;

    cout << '2. Display a priorty Queue'
         << endl;

    cout << '3. Delete an element'
         << endl;

    cout << '4. Search an element'
         << endl;

    cout << '5. Exit'
         << endl;

    cin >> choice;

    switch(choice)

    {

    case 1:

        cout << 'Enter a Data to enter Queue'
             << endl;

        cin >> DT;

        PQ.Insert(DT);

        break;

    case 2:

        PQ.Display();

        break;

    case 3:

        {
```

```
        int choice;

        cout << 'Enter your choice'
             << endl;

        cout << '1. Maximum Priority Queue'
             << endl;

        cout << '2. Minimum Priority Queue'
             << endl;

        cin >> choice;

        switch(choice)

        {

        case 1:

            PQ.Delete(PQ.Maximum());

            break;

        case 2:

            PQ.Delete(PQ.Minimum());

            break;

        default:

            cout << 'Sorry Not a correct choice'
                 << endl;

        }

    }

    break;

case 4:

    cout << 'Enter a Data to Search in Queue'
         << endl;

    cin >> DT;

    if(PQ.Search(DT) != FALSE){

        cout << DT
             << ' Is present in Queue'
             << endl;
```

```
            }else{

                cout << DT
                     << ' is Not present in Queue'
                     << endl;

            }
            break;

        case 5:

            exit(0);

        default:

            cout << 'Cannot process your choice'
                 << endl;

        }

    }

}

*/
#endif
```

PriorityQueue.cc:

```
#ifndef PRIORITY_QUEUE_H_

#define PRIORITY_QUEUE_H_

#include <iostream>

using namespace std;

#include <iostream>

#include <cstdlib>

enum{

 FALSE = 0,

 TRUE = -1

};

////////////////////////////////

///// Implements Priority Queue
```

```
////////////////////////////////

// Class Prioriry Queue

class PriorityQueue{

 private:

  // Node of Priority Queue

  struct Node{

        struct Node *Previous;

        int Data;

        struct Node *Next;

  }Current;

  struct Node *head;  // Pointer to Head

  struct Node *ptr;

        // Pointer for travelling through Queue

  static int NumOfNodes;

        // Keeps track of Number of nodes

 public:

  PriorityQueue(void);

  int Maximum(void);

  int Minimum(void);

  void Insert(int);

  int Delete(int);

  void Display(void);

  int Search (int);

  ~PriorityQueue(void);

};

/*

//Main Function
```

```
void main()

{

    PriorityQueue PQ;

    int choice;

    int DT;

    while(1)

    {

        cout <<'Enter your choice'
             << endl;

        cout << '1. Insert an element'
             << endl;

        cout << '2. Display a priorty Queue'
             << endl;

        cout << '3. Delete an element'
             << endl;

        cout << '4. Search an element'
             << endl;

        cout << '5. Exit'
             << endl;

        cin >> choice;

        switch(choice)

        {

        case 1:

            cout << 'Enter a Data to enter Queue'
                 << endl;

            cin >> DT;

            PQ.Insert(DT);

            break;

        case 2:

            PQ.Display();
```

```
    break;

case 3:

    {

        int choice;

        cout << 'Enter your choice'
             << endl;

        cout << '1. Maximum Priority Queue'
             << endl;

        cout << '2. Minimum Priority Queue'
             << endl;

        cin >> choice;

        switch(choice)

        {

        case 1:

            PQ.Delete(PQ.Maximum());

            break;

        case 2:

            PQ.Delete(PQ.Minimum());

            break;

        default:

            cout << 'Sorry Not a correct choice'
                 << endl;

        }

    }

    break;

case 4:

    cout << 'Enter a Data to Search in Queue'
         << endl;

    cin >> DT;
```

```
            if(PQ.Search(DT) != FALSE){

                cout << DT
                     << ' Is present in Queue'
                     << endl;

            }else{

                cout << DT
                     << ' is Not present in Queue'
                     << endl;

            break;

        case 5:

            exit(0);

        default:

            cout << 'Cannot process your choice'
                 << endl;

        }

    }

}

*/

#endif
```

If you encounter any problems or errors, please let me know by providing an example of the code, input, output, and an explanation. Thanks.