ST. CHARLES COMMUNITY COLLEGE

CPT-281 - Introduction to Data Structures with C++

Module 6

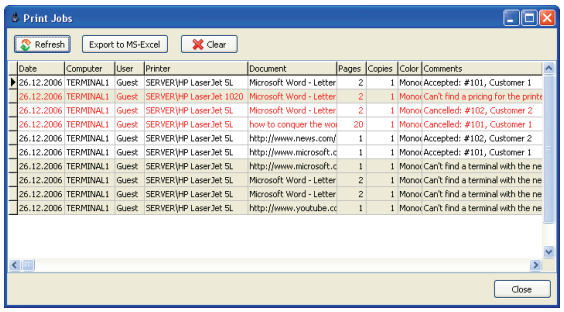
Queues, Deques

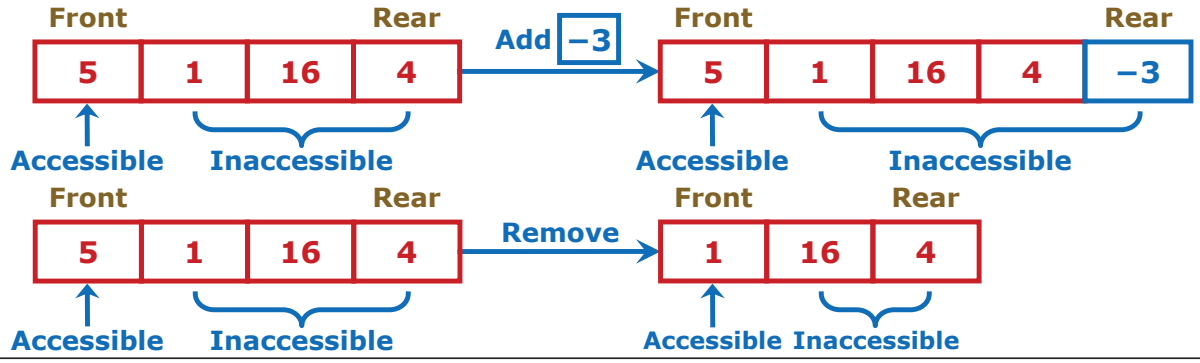
Dayu Wang

- **Queues**

→ **Queues are abstract data structures (ADT) with the property that only the front element is accessible.**

Example 1	Example 2
Waiting line	Print jobs





- **Class-Member Functions of Queues**

  → **Theoretically, queues only support 6 class-member functions.**

  You can only use the 6 functions below when you use queues.

Functions	Behavior
`size_t size() const;`	Returns the number of elements in the queue.
`bool empty() const;`	Tests whether the queue is empty.
`T& front();`	Returns the element at the front end of the queue (l-value).
`const T& front() const;`	Returns the element at the front end of the queue (r-value).
`void pop();`	Deletes an element from the front end of the queue.
`void push(const T&);`	Adds an element to the rear end of the queue.

- **Using linked list to implement queue**

  → **Singly-linked list or doubly-linked list?**

  Singly-linked list

  → **In class data fields, do we need to keep reference of front node only or both front and rear nodes?**

  Both front and rear nodes

→ **The `size()` function**

```
1  /** Returns the number of elements in the queue.
2      @return: number of elements in the queue
3  */
4  template<class T>
5  size_t List_Queue<T>::size() const { return num_of_items; }
6  // Time complexity: O(1)
```

→ **The `empty()` function**

```
1  /** Tests whether the queue is empty.
2      @return: {true} if the queue is empty; {false} otherwise
3  */
4  template<class T>
5  bool List_Queue<T>::empty() const { return !size(); }
6  // Time complexity: O(1)
```

→ The front() function (l-value)

```
1  /** Returns the element at the front end of the queue (l-value).
2      @return: element at the front end of the queue (l-value)
3      @throws exception: queue is empty.
4  */
5  template<class T>
6  T& List_Queue<T>::front() {
7      if (empty()) { throw exception("Accessing empty queue"); }
8      return front_node->data;
9  }  // Time complexity: O(1)
```

→ The front() function (r-value)

```
1  /** Returns the element at the front end of the queue (r-value).
2      @return: element at the front end of the queue (r-value)
3      @throws exception: queue is empty.
4  */
5  template<class T>
6  const T& List_Queue<T>::front() const {
7      if (empty()) { throw exception("Accessing empty queue"); }
8      return front_node->data;
9  }  // Time complexity: O(1)
```


→ The pop() function

```
1  /** Deletes an element from the front end of the queue.
2      @throws exception: queue is empty.
3  */
4  template<class T>
5  void List_Queue<T>::pop() {
6      if (empty()) { throw exception("Accessing empty queue"); }
7      Node* to_be_deleted = front_node;
8      if (num_of_items-- == 1) { front_node = rear_node = NULL; }
9      else { front_node = front_node->next; }
10     delete to_be_deleted;
11 }  // Time complexity: O(1)
```

→ The push() function

```
1  /** Adds an element to the rear end of the queue.
2      @param item: element to add to the queue
3  */
4  template<class T>
5  void List_Queue<T>::push(const T& item) {
6      if (!(num_of_items++)) { front_node = rear_node = new Node(item); }
7      else {
8          rear_node->next = new Node(item);
9          rear_node = rear_node->next;
10     }
11 }  // Time complexity: O(1)
```

- **Using circular array to implement queue**

→ **Although the time efficiency of using a linked list to implement the queue is acceptable, there is some space inefficiency.**

Each node of a singly-linked list contains a reference to its successor.

These additional references will increase the storage space required.

→ **Can we use an array to implement queue?**

- **We can do an insertion at the rear of the array in $O(1)$ time.**
- **However, a removal from the front will be an $O(n)$ process if we shift all the elements that follow the first one over to fill the space vacated.**

→ **How to avoid this inefficiency?**

We can use a circular array.

In addition to the `capacity` and `num_of_items`, we need two additional integer data fields, `front_index` and `rear_index`, to keep track of the indices to the queue elements at the front and rear of the queue.

```
1  // Data fields
2  T* data;  // Stores the elements in the queue.
3  static const size_t DEFAULT_CAPACITY;  // Default capacity
4  size_t capacity;  // Stores the maximum capacity of the queue.
5  size_t num_of_items;  // Stores the number of elements in the queue.
6  int front_index;  // Stores the index of the front end of the queue.
7  int rear_index;  // Stores the index of the rear end of the queue.
```

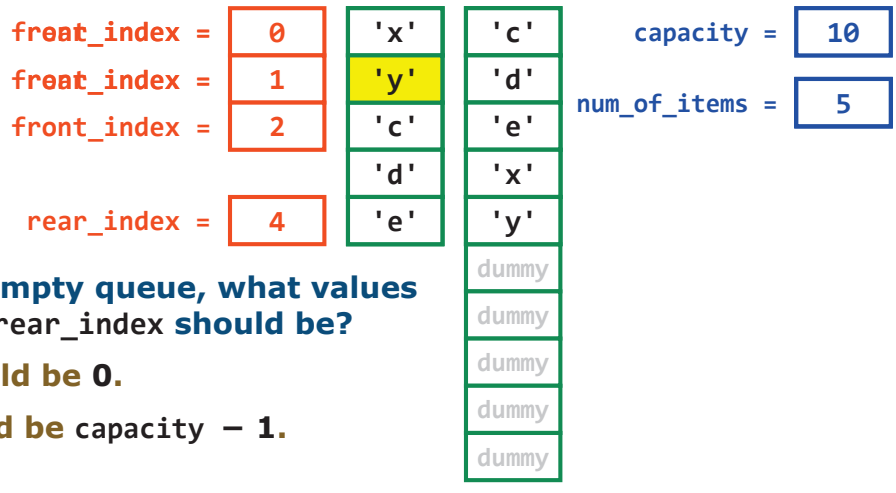

→ **How to implement the 6 functions supported by queue?**

The `.size()`, `.empty()`, and `.front()` functions can be implemented straightforward, since the queue itself is unmodified.

What happens to the queue if `.pop()` and `.push()` functions are called?

[Example]

- **Call `.pop()`**
- **Call `.pop()`**
- **Call `.push('x')`**
- **Call `.push('y')`**
- **Call `.resize()`**



→ **Initially, for an empty queue, what values `front_index` and `rear_index` should be?**

`front_index` should be 0.

`rear_index` should be `capacity` − 1.

- **Using circular array to implement queue**

→ **The pop() function**

```
1  /** Deletes an element from the front end of the queue.
2      @throws exception: queue is empty.
3  */
4  template<class T>
5  void Circular_Array_Queue<T>::pop() {
6      if (empty()) { throw exception("Accessing empty queue"); }
7      front_index = (front_index + 1) % capacity;
8      num_of_items--;
9  }  // Time complexity: O(1)
```

→ **The push() function**

```
1  /** Adds an element to the rear end of the queue.
2      @param item: element to add to the queue
3  */
4  template<class T>
5  void Circular_Array_Queue<T>::push(const T& item) {
6      if (num_of_items == capacity) { resize(); }
7      rear_index = (rear_index + 1) % capacity;
8      data[rear_index] = item;
9      num_of_items++;
10 }  // Time complexity: O(1)
```


- **Application of Queues**

→ **Scheduling**

Queue implements a first-in-first-out mechanism.

→ **Breadth-First-Search (BFS)**

Queue is an intermediate data structure.

- **Deques**

  → "Deque" is pronounced like "deck".

  → A deque (double-ended queue) is a data structure that combines the features of a stack and a queue.

  → With a deque, you can insert, access, and remove items at either end.

- **Deque Functions**

Functions	Behavior
`size_t size() const;`	Returns the number of elements in the deque.
`bool empty() const;`	Tests whether the deque is empty.
`T& front();`	Returns the element at front end of the deque (l-value).
`const T& front() const;`	Returns the element at front end of the deque (r-value).
`T& back();`	Returns the element at rear end of the deque (l-value).
`const T& back() const;`	Returns the element at rear end of the deque (r-value).
`void pop_front();`	Deletes an element from the front end of the deque.
`void pop_back();`	Deletes an element from the rear end of the deque.
`void push_front(const T&);`	Adds an element to the front end of the deque.
`void push_back(const T&);`	Adds an element to the rear end of the deque.

- **Using linked list to implement deque**

  → Singly-linked list or doubly-linked list?

  Doubly-linked list

  → In class data fields, do we need to keep reference of front node only or front and rear nodes?

  Front and rear nodes

→ The size() function

```
1  /** Returns the number of elements in the deque.
2      @return: number of elements in the deque
3  */
4  template<class T>
5  size_t List_Deque<T>::size() const { return num_of_items; }
6  // Time complexity: O(1)
```

→ The empty() function

```
1  /** Tests whether the deque is empty
2      @return: {true} if the deque is empty; {false} otherwise
3  */
4  template<class T>
5  bool List_Deque<T>::empty() const { return !size(); }
6  // Time complexity: O(1)
```


→ The front() function (l-value)

```
1  /** Returns the element at the front end of the deque (l-value).
2      @return: element at the front end of the deque (l-value)
3      @throws exception: deque is empty.
4  */
5  template<class T>
6  T& List_Deque<T>::front() {
7      if (empty()) { throw exception("Accessing empty deque"); }
8      return front_node->data;
9  }  // Time complexity: O(1)
```

→ The front() function (r-value)

```
1  /** Returns the element at the front end of the deque (r-value).
2      @return: element at the front end of the deque (r-value)
3      @throws exception: deque is empty.
4  */
5  template<class T>
6  const T& List_Deque<T>::front() const {
7      if (empty()) { throw exception("Accessing empty deque"); }
8      return front_node->data;
9  }  // Time complexity: O(1)
```

→ The back() function (l-value)

```
1  /** Returns the element at the rear end of the deque (l-value).
2      @return: element at the rear end of the deque (l-value)
3      @throws exception: deque is empty.
4  */
5  template<class T>
6  T& List_Deque<T>::back() {
7      if (empty()) { throw exception("Accessing empty deque"); }
8      return rear_node->data;
9  }  // Time complexity: O(1)
```

→ The back() function (r-value)

```
1  /** Returns the element at the rear end of the deque (r-value).
2      @return: element at the rear end of the deque (r-value)
3      @throws exception: deque is empty.
4  */
5  template<class T>
6  const T& List_Deque<T>::back() const {
7      if (empty()) { throw exception("Accessing empty deque"); }
8      return rear_node->data;
9  }  // Time complexity: O(1)
```


→ The pop_front() function

```
1  /** Deletes an element from the front end of the deque.
2      @throws exception: deque is empty.
3  */
4  template<class T>
5  void List_Deque<T>::pop_front() {
6      if (empty()) { throw exception("Accessing empty deque"); }
7      D_Node* to_be_deleted = front_node;
8      if (num_of_items-- == 1) { front_node = rear_node = NULL; }
9      else {
10         front_node = front_node->next;
11         front_node->prev = NULL;
12     }
13     delete to_be_deleted;
14 }  // Time complexity: O(1)
```

→ The pop_back() function

```
1   /** Deletes an element from the rear end of the deque.
2       @throws exception: deque is empty.
3   */
4   template<class T>
5   void List_Deque<T>::pop_back() {
6       if (empty()) { throw exception("Accessing empty deque"); }
7       D_Node* to_be_deleted = rear_node;
8       if (num_of_items-- == 1) { front_node = rear_node = NULL; }
9       else {
10          rear_node = rear_node->prev;
11          rear_node->next = NULL;
12      }
13      delete to_be_deleted;
14  }  // Time complexity: O(1)
```


→ The push_front() function

```
1   /** Adds an element to the front end of the deque.
2       @param item: element to add to the deque
3   */
4   template<class T>
5   void List_Deque<T>::push_front(const T& item) {
6       if (!(num_of_items++)) { front_node = rear_node = new D_Node(item); }
7       else {
8           front_node->prev = new D_Node(item);
9           front_node->prev->next = front_node;
10          front_node = front_node->prev;
11      }
12  }  // Time complexity: O(1)
```

→ **The** `push_back()` **function**

```
1  /** Adds an element to the rear end of the deque.
2      @param item: element to add to the deque
3  */
4  template<class T>
5  void List_Deque<T>::push_back(const T& item) {
6      if (!(num_of_items++)) { front_node = rear_node = new D_Node(item); }
7      else {
8          rear_node->next = new D_Node(item);
9          rear_node->next->prev = rear_node;
10         rear_node = rear_node->next;
11     }
12 }  // Time complexity: O(1)
```


- **Using circular array to implement deque**

  → `.pop_front()` ➡ **Circularly increment** `front_index`**.**

  → `.pop_back()` ➡ **Circularly decrement** `rear_index`**.**

  → `.push_front()` ➡ **Circularly decrement** `front_index`**.**

  → `.push_back()` ➡ **Circularly increment** `rear_index`**.**

- **Using circular array to implement deque**

→ **The pop_front() function**

```
1  /** Deletes an element from the front end of the deque.
2      @throws exception: deque is empty.
3  */
4  template<class T>
5  void Circular_Array_Deque<T>::pop_front() {
6      if (empty()) { throw exception("Accessing empty deque"); }
7      front_index = (front_index + 1) % capacity;
8      num_of_items--;
9  }  // Time complexity: O(1)
```

→ **The pop_back() function**

```
1  /** Deletes an element from the rear end of the deque.
2      @throws exception: deque is empty.
3  */
4  template<class T>
5  void Circular_Array_Deque<T>::pop_back() {
6      if (empty()) { throw exception("Accessing empty deque"); }
7      rear_index = (rear_index - 1 + capacity) % capacity;
8      num_of_items--;
9  }  // Time complexity: O(1)
```


→ **The push_front() function**

```
1  /** Adds an element to the front end of the deque.
2      @param item: element to add to the deque
3  */
4  template<class T>
5  void Circular_Array_Deque<T>::push_front(const T& item) {
6      if (num_of_items == capacity) { resize(); }
7      front_index = (front_index - 1 + capacity) % capacity;
8      data[front_index] = item;
9      num_of_items++;
   }  // Time complexity: O(1)
```

→ **The push_back() function**

```
1  /** Adds an element to the rear end of the deque.
2      @param item: element to add to the deque
3  */
4  template<class T>
5  void Circular_Array_Deque<T>::push_back(const T& item) {
6      if (num_of_items == capacity) { resize(); }
7      rear_index = (rear_index + 1) % capacity;
8      data[rear_index] = item;
9      num_of_items++;
10 }  // Time complexity: O(1)
```

- **Priority Queues**

  → **Priority queues are not ~~sequential data structures~~.**

  → **We will discuss priority queues in future lectures.**

- **C++ build-in data structures**

  → **Libraries to include:**

```
1  #include <stack>
2  #include <queue>
3  #include <deque>
```

  → **All the classes are template classes.**

```
1  stack<int> stk;
2  queue<string> que;
3  deque<double> deq;
```

- **We completed sequential data structures.**

Vector	Linked List	Stack	Queue	Deque
Index/Iterator	Iterator	Only top element	Only front element	Front and rear elements
Basic	Basic	Implemented by Vector Linked List	Implemented by Circular Array Linked List	Implemented by Circular Array Linked List