MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

19 Queues

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

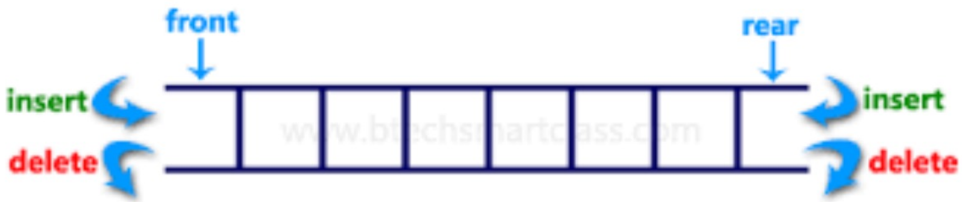
March 7, 2022

Admin.

- One more day for the assignment 2!

Double-Ended Queues

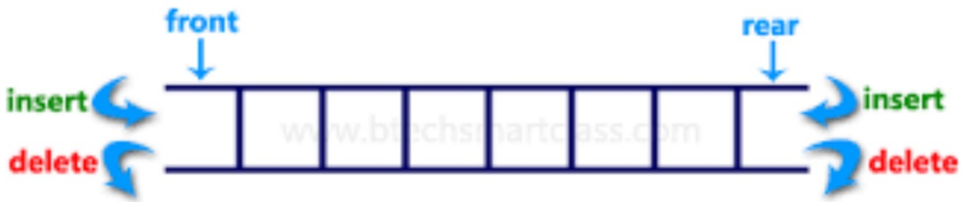
- Double-Ended Queues (sometimes pronounced like ”deck”)
  - supports insertion and deletion at both the front and the rear of the queue



- **insertFront(e):** Insert a new element e at the beginning of the deque.
- **insertBack(e):** Insert a new element e at the end of the deque.
- **eraseFront():** Remove the first element of the deque; an error occurs if the deque is empty.
- **eraseBack():** Remove the last element of the deque; an error occurs if the deque is empty.
- **front():** Return the first element of the deque; an error occurs if the deque is empty.
- **back():** Return the last element of the deque; an error occurs if the deque is empty.
- **size():** Return the number of elements of the deque.
- **empty():** Return true if the deque is empty and false otherwise.

Double-Ended Queues

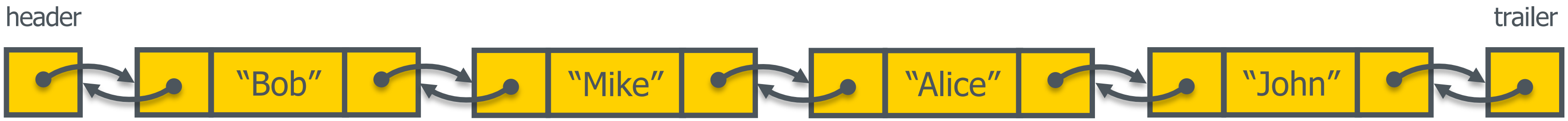
- Double-Ended Queues (sometimes pronounced like ”deck”)
  - supports insertion and deletion at both the front and the rear of the queue



- A running example:

Operation	*Output*	*D*
insertFront(3)	–	(3)
insertFront(5)	–	(5,3)
front()	5	(5,3)
eraseFront()	–	(3)
insertBack(7)	–	(3,7)
back()	7	(3,7)
eraseFront()	–	(7)
eraseBack()	–	()

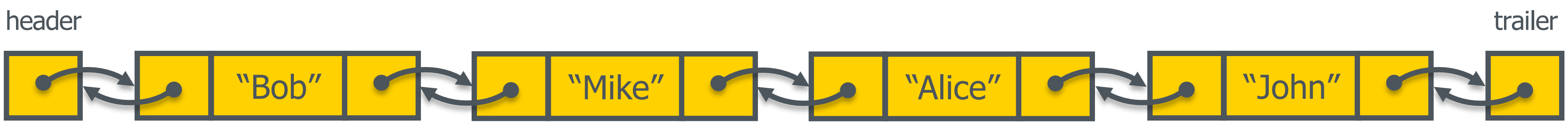
Implementation with Doubly Linked List



- *We will use the functionalities provided by the DLL to implement LinkedDeque's functions. We have seen this pattern a few times before.*

```
typedef string Elem;                          // deque element type
class LinkedDeque {                           // deque as doubly linked list
public:
  LinkedDeque();                              // constructor
  int size() const;                           // number of items in the deque
  bool empty() const;                         // is the deque empty?
  const Elem& front() const throw(DequeEmpty); // the first element
  const Elem& back() const throw(DequeEmpty); // the last element
  void insertFront(const Elem& e);            // insert new first element
  void insertBack(const Elem& e);             // insert new last element
  void removeFront() throw(DequeEmpty);       // remove first element
  void removeBack() throw(DequeEmpty);        // remove last element
private:                                      // member data
  DLinkedList D;                              // linked list of elements
  int n;                                      // number of elements
};
```

Implementation with Doubly Linked List



- We will use the functionalities provided by the DLL to implement LinkedDeque's functions. We have seen this pattern a few times before.
- Performance of a deque realized by a doubly linked list.

Operation	***Time***
size	$O(1)$
empty	$O(1)$
front, back	$O(1)$
insertFront, insertBack	$O(1)$
eraseFront, eraseBack	$O(1)$

- The space used usage is O(n)

```
                                            // insert new first element
void LinkedDeque::insertFront(const Elem& e) {
  D.addFront(e);
  n++;
}
                                            // insert new last element
void LinkedDeque::insertBack(const Elem& e) {
  D.addBack(e);
  n++;
}
                                            // remove first element
void LinkedDeque::removeFront() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeFront of empty deque");
  D.removeFront();
  n--;
}
                                            // remove last element
void LinkedDeque::removeBack() throw(DequeEmpty) {
  if (empty())
    throw DequeEmpty("removeBack of empty deque");
  D.removeBack();
  n--;
}
```

Adapter Design Pattern

- Design pattern: which describes a solution to a “typical” software design problem.
  - provides a general template for a solution that can be applied in many different situations.
  - describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand.
  - In Algorithms:
    - Recursion
    - Using Stack to solve problems
  - In Software Engineering
    - Adapter pattern
    - Iterator pattern
- You remember from previous LinkedDeque, and also Circular Linked List-based implementation of Queue

Adapter Design Pattern

- You remember from previous LinkedDeque, and also Circular Linked List-based implementation of Queue that we took an existing data structure and **adapted** it
  - E.g. we added size **n**
  - We added operations that are meaningful for the new data structure
- For the operations, we have simply **mapped** each deque operation to the corresponding operation of DLinkedList.
- An adapter (also called a wrapper) is a data structure that translates one interface to another.
  - e.g.: In the LinkedDeque implementation:
    - deque operation **insertFront** is mapped to the corresponding operation of DLinkedList **addFront**

Adapter Design Pattern

- Implementing a Stack using Deque:

```
typedef string Elem;                          // element type
class DequeStack {                            // stack as a deque
public:
  DequeStack();                               // constructor
  int size() const;                           // number of elements
  bool empty() const;                         // is the stack empty?
  const Elem& top() const throw(StackEmpty); // the top element
  void push(const Elem& e);                   // push element onto stack
  void pop() throw(StackEmpty);               // pop the stack
private:
  LinkedDeque D;                              // deque of elements
};
```

Stack Method	*Deque Implementation*
size()	size()
empty()	empty()
top()	front()
push(o)	insertFront(o)
pop()	eraseFront()

```
DequeStack::DequeStack()                      // constructor
  : D() { }
                                              // number of elements
int DequeStack::size() const
  { return D.size(); }
                                              // is the stack empty?
bool DequeStack::empty() const
  { return D.empty(); }
                                              // the top element
const Elem& DequeStack::top() const throw(StackEmpty) {
  if (empty())
    throw StackEmpty("top of empty stack");
  return D.front();
}
                                              // push element onto stack
void DequeStack::push(const Elem& e)
  { D.insertFront(e); }
                                              // pop the stack
void DequeStack::pop() throw(StackEmpty)
{
  if (empty())
    throw StackEmpty("pop of empty stack");
  D.removeFront();
}
```

Adapter Design Pattern

- Implementing a Queue using Deque:

Queue Method	*Deque Implementation*
size()	size()
empty()	empty()
front()	front()
enqueue(e)	insertBack(e)
dequeue()	eraseFront()

- The operations are equally efficient.
- We have used and will use this design pattern many times.

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - string (String class with all operations)
  - stack (Container with last-in, first-out access)
  - queue (Container with first-in, first-out access)
  - deque (Double-ended queue)
  - vector (Resizable array)
  - list (Doubly linked list)
  - priority queue (Queue ordered by value)
  - set (Set)
  - map Associative array (dictionary)

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - string:

s.find(p)	Return the index of first occurrence of string *p* in *s*
s.find(p, i)	Return the index of first occurrence of string *p* in *s* on or after position *i*
s.substr(i,m)	Return the substring starting at position *i* of *s* and consisting of *m* characters
s.insert(i, p)	Insert string *p* just prior to index *i* in *s*
s.erase(i, m)	Remove the substring of length *m* starting at index *i*
s.replace(i, m, p)	Replace the substring of length *m* starting at index *i* with *p*
getline(is, s)	Read a single line from the input stream *is* and store the result in *s*

```
#include <string>
using std::string;
// ...
string s = "to be";
string t = "not " + s;             // t = "not to be"
string u = s + " or " + t;         // u = "to be or not to be"
if (s > t)                         // true: "to be" > "not to be"
  cout << u;                       // outputs "to be or not to be"
```

```
string s = "John";                 // s = "John"
int i = s.size();                  // i = 4
char c = s[3];                     // c = 'n'
s += " Smith";                     // now s = "John Smith"
```

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - vector:

```
#include <vector>
using namespace std;                     // make std accessible

vector<int> scores(100);                 // 100 integer scores
vector<char> buffer(500);                // buffer of 500 characters
vector<Passenger> passenList(20);        // list of 20 Passengers
```

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - stack:

```
#include <stack>
using std::stack;                    // make stack accessible
stack<int> myStack;                  // a stack of integers
```

size():	Return the number of elements in the stack.
empty():	Return true if the stack is empty and false otherwise.
push(e):	Push e onto the top of the stack.
pop():	Pop the element at the top of the stack.
top():	Return a reference to the element at the top of the stack.

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - queue:

```
#include <queue>
using std::queue;                    // make queue accessible
queue<float> myQueue;                // a queue of floats
```

size():	Return the number of elements in the queue.
empty():	Return true if the queue is empty and false otherwise.
push(e):	Enqueue e at the rear of the queue.
pop():	Dequeue the element at the front of the queue.
front():	Return a reference to the element at the queue's front.
back():	Return a reference to the element at the queue's rear.

Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of classes for common data structures. In addition to the string class, which we have seen many times, it also provides data structures for the following standard containers.
  - deque:

```
#include <deque>
using std::deque;                    // make deque accessible
deque<string> myDeque;               // a deque of strings
```

size():	Return the number of elements in the deque.
empty():	Return true if the deque is empty and false otherwise.
push_front(e):	Insert e at the beginning the deque.
push_back(e):	Insert e at the end of the deque.
pop_front():	Remove the first element of the deque.
pop_back():	Remove the last element of the deque.
front():	Return a reference to the deque's first element.
back():	Return a reference to the deque's last element.

List and Sequence Containers

- Vector (also called Array List)
  - Access each element using a notion of index in [0,n-1]
  - Index of element e: the number of elements that are before e
  - Typically we use the "index" (e.g., [ ])
  - A more general ADT than "array"
- List
  - Not using an index to access, but use a node to access
  - Insert a new element e before some "position" p
  - A more general ADT than "linked list"
- Sequence
  - Can access an element as vector and list (using both index and position)

The Vector ADT

- The Vector or Array List ADT extends the notion of array by storing a sequence of objects
- An element can be accessed, inserted or removed by specifying its index (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)
- Main methods:

at(i): Return the element of V with index i; an error condition occurs if i is out of range.

set(i, e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i, e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation **at(i)** is implemented in O(1) time by returning A[i]
- Operation set(i,o) is implemented in O(1) time by performing A[i] = o

at(i): Return the element of V with index i; an error condition occurs if i is out of range.

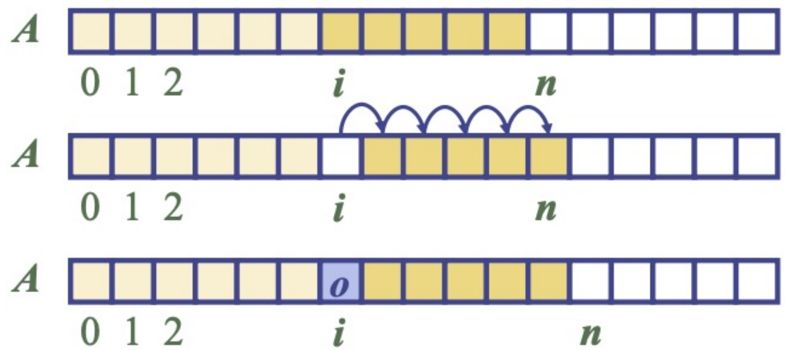
set(i,e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i,e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Insertion

- In operation insert(i, o), we need to make room for the new element by shifting forward the n - i elements A[i], …, A[n - 1]
  - In the worst case (i = 0), this takes O(n) time



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

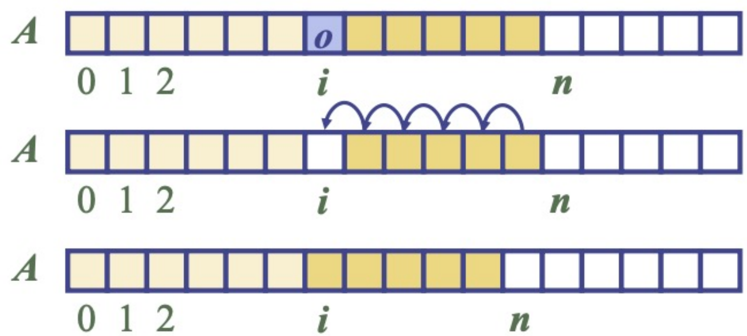
set(i,e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i,e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Removal

- In operation erase(i), we need to fill the hole left by the removed element by shifting backward the n - i - 1 elements A[i + 1], …, A[n - 1]
  - In the worst case (i = 0), this takes O(n) time



at(i): Return the element of V with index i; an error condition occurs if i is out of range.

set(i,e): Replace the element at index i with e; an error condition occurs if i is out of range.

insert(i,e): Insert a new element e into V to have index i; an error condition occurs if i is out of range.

erase(i): Remove from V the element at index i; an error condition occurs if i is out of range.

Array-based Implementation of Vector - Performance

- In the array-based implementation of an array list:
  - The space used by the data structure is O(n)
  - size, empty, at and set run in O(1) time
  - insert and erase run in O(n) time in worst case
- If we use the array in a circular fashion, operations insert(0, x) and erase(0, x) run in O(1) time
- In an insert operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

Operation	*Time*
size()	$O(1)$
empty()	$O(1)$
at(i)	$O(1)$
set(i,e)	$O(1)$
insert(i,e)	$O(n)$
erase(i)	$O(n)$

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n insert(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an insert operation the average time taken by an insert over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- We replace the array $k = n/c$ times
- The total time $T(n)$ of a series of n insert operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc =$$

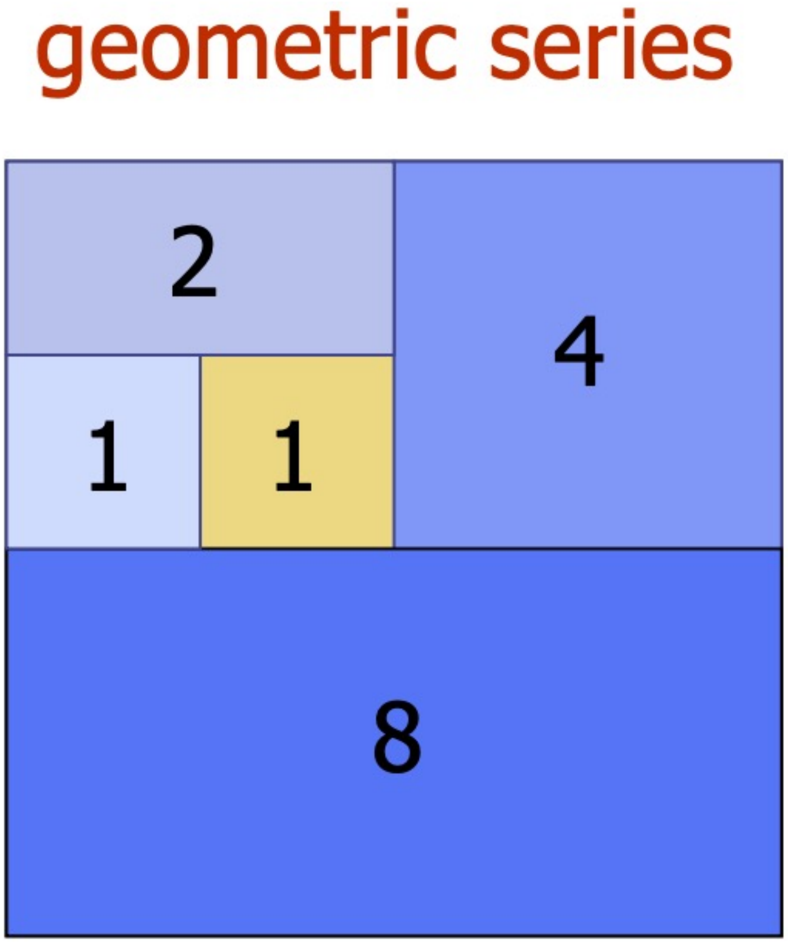
$$n + c(1 + 2 + 3 + \ldots + k) =$$

$$n + ck(k + 1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert operation is $O(n)$

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n insert operations is proportional to
- $n + 1 + 2 + 4 + 8 + \ldots + 2^k =$
  $n + 2^{k+1} - 1 =$
- $3n - 1$
- $T(n)$ is $O(n)$
- The amortized time of an insert operation is $O(1)$



Questions?