CH5. STACKS, QUEUES, DEQUES

CSED233 Data Structure

Prof. Hwanjo Yu
POSTECH

Stacks (§ 5.1)

- Stacks
  - *Last-in-first-out (LIFO) data structure*
  - *"push" and "pop" operations*
  - *E.g., do and undo in text editors*
  - *Member functions*
    - push(e)
    - pop()
    - top()
    - size()
    - empty()
  - *Every operation is* $O(1)$.

Operation	*Output*	*Stack Contents*
push(5)	–	(5)
push(3)	–	(5,3)
pop()	–	(5)
push(7)	–	(5,7)
pop()	–	(5)
top()	*5*	(5)
pop()	–	()
pop()	*"error"*	()
top()	*"error"*	()
empty()	*true*	()
push(9)	–	(9)
push(7)	–	(9,7)
push(3)	–	(9,7,3)
push(5)	–	(9,7,3,5)
size()	*4*	(9,7,3,5)
pop()	–	(9,7,3)
push(8)	–	(9,7,3,8)
pop()	–	(9,7,3)
top()	*3*	(9,7,3)

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };          // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);   // constructor from capacity
  int size() const;                     // number of items in the stack
  bool empty() const;                   // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);
  // ...housekeeping functions omitted
private:
  E* S;
  int capacity;
  int t;
};
```

Code Fragment 5.4: The class ArrayStack, wi

```
template <typename E> ArrayStack<E>::ArrayStack(int cap)
  : S(new E[cap]), capacity(cap), t(-1) { } // constructor from capacity

template <typename E> int ArrayStack<E>::size() const
  { return (t + 1); }                   // number of items in the stack

template <typename E> bool ArrayStack<E>::empty() const
  { return (t < 0); }                   // is the stack empty?

template <typename E>                   // return top of stack
const E& ArrayStack<E>::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S[t];
}

template <typename E>                   // push element onto the stack
void ArrayStack<E>::push(const E& e) throw(StackFull) {
  if (size() == capacity) throw StackFull("Push to full stack");
  S[++t] = e;
}

template <typename E>                   // pop the stack
void ArrayStack<E>::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --t;
}
```

Code Fragment 5.5: Implementations of the member functions of class ArrayStack (excluding housekeeping functions).

```
template <typename E>
class ArrayStack {
  enum { DEF_CAPACITY = 100 };          // default stack capacity
public:
  ArrayStack(int cap = DEF_CAPACITY);   // constructor from capacity
  int size() const;                     // number of items in the stack
  bool empty() const;                   // is the stack empty?
  const E& top() const throw(StackEmpty); // get the top element
  void push(const E& e) throw(StackFull); // push element onto stack
  void pop() throw(StackEmpty);         // pop the stack
  // ...housekeeping functions omitted
private:                                // member data
  E* S;                                 // array of stack elements
  int capacity;                         // stack capacity
  int t;                                // index of the top of the stack
};
```

Code Fragment 5.4: The class ArrayStack, which implements the Stack interface.

```
ArrayStack<int> A;                      // A = [], size = 0
A.push(7);                              // A = [7*], size = 1
A.push(13);                             // A = [7, 13*], size = 2
cout << A.top() << endl; A.pop();       // A = [7*], outputs: 13
A.push(9);                              // A = [7, 9*], size = 2
cout << A.top() << endl;                // A = [7, 9*], outputs: 9
cout << A.top() << endl; A.pop();       // A = [7*], outputs: 9
ArrayStack<string> B(10);               // B = [], size = 0
B.push("Bob");                          // B = [Bob*], size = 1
B.push("Alice");                        // B = [Bob, Alice*], size = 2
cout << B.top() << endl; B.pop();       // B = [Bob*], outputs: Alice
B.push("Eve");                          // B = [Bob, Eve*], size = 2
```

Code Fragment 5.6: An example of the use of the ArrayStack class. The contents of the stack are shown in the comment following the operation. The top of the stack is indicated by an asterisk ("*").

```
typedef string Elem;                          // stack element type
class LinkedStack {                           // stack as a linked list
public:
  LinkedStack();                              // constructor
  int size() const;                           // number of items in the stack
  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:                                      // member data
  SLinkedList<Elem> S;                        // linked list of elements
  int n;                                      // number of elements
};
```

Code Fragment 5.7: The class LinkedStack, a linked list implementation of a stack.

```
                                              // get the top element
const Elem& LinkedStack::top() const throw(StackEmpty) {
  if (empty()) throw StackEmpty("Top of empty stack");
  return S.front();
}
void LinkedStack::push(const Elem& e) {    // push element onto stack
  ++n;
  S.addFront(e);
}
                                              // pop the stack
void LinkedStack::pop() throw(StackEmpty) {
  if (empty()) throw StackEmpty("Pop from empty stack");
  --n;
  S.removeFront();
}
```

Code Fragment 5.9: Principal operations for the LinkedStack class.

```
template <typename E>
void reverse(vector<E>& V) {                    // reverse a vector
  ArrayStack<E> S(V.size());
  for (int i = 0; i < V.size(); i++)            // push elements onto stack
    S.push(V[i]);
  for (int i = 0; i < V.size(); i++) {          // pop them in reverse order
    V[i] = S.top();  S.pop();
  }
}
```

Code Fragment 5.10: A generic function that uses a stack to reverse a vector.

Algorithm ParenMatch(X,n):

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in X match

```
Let S be an empty stack
for i ← 0 to n − 1 do
  if X[i] is an opening grouping symbol then
    S.push(X[i])
  else if X[i] is a closing grouping symbol then
    if S.empty() then
      return false          {nothing to match with}
    if S.top() does not match the type of X[i] then
      return false          {wrong type}
    S.pop()
if S.empty() then
  return true               {every symbol matched}
else
  return false              {some symbols were never matched}
```

Code Fragment 5.11: Algorithm for matching grouping symbols in an arithmetic expression.

Queues (§ 5.2)

- Queues
  - *First-in-first-out (FIFO) data structure*
  - *"enqueue" and "dequeue" operations*
  - *Member functions (STL has different names)*
    - enqueue(e)
    - dequeue()
    - front()
    - size()
    - empty()
  - *Every operation is O(1).*

Operation	Output	front ← Q ← rear
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
front()	5	(5, 3)
size()	2	(5, 3)
dequeue()	–	(3)
enqueue(7)	–	(3, 7)
dequeue()	–	(7)
front()	7	(7)
dequeue()	–	()
dequeue()	"error"	()
empty()	true	()

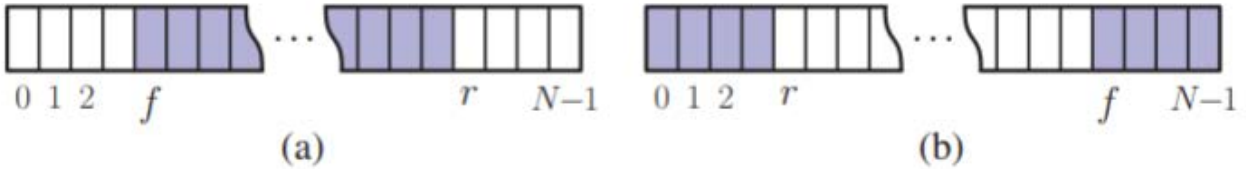

Figure 5.4: Using array Q in a circular fashion: (a) the "normal" configuration with $f \leq r$; (b) the "wrapped around" configuration with $r < f$. The cells storing queue elements are shaded.

```
Algorithm size():
    return n
Algorithm empty():
    return (n = 0)
Algorithm front():
    if empty() then
        throw QueueEmpty exception
    return Q[f]
Algorithm dequeue():
    if empty() then
        throw QueueEmpty exception
    f ← (f + 1) mod N
    n = n − 1
Algorithm enqueue(e):
    if size() = N then
        throw QueueFull exception
    Q[r] ← e
    r ← (r + 1) mod N
    n = n + 1
```

Code Fragment 5.17: Implementation of a queue using a circular array.

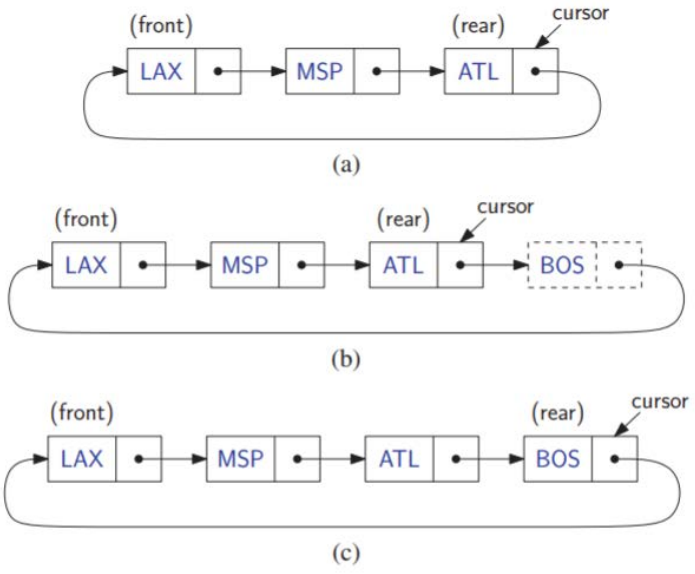

Figure 5.5: Enqueueing "BOS" into a queue represented as a circularly linked list: (a) before the operation; (b) after adding the new node; (c) after advancing the cursor.

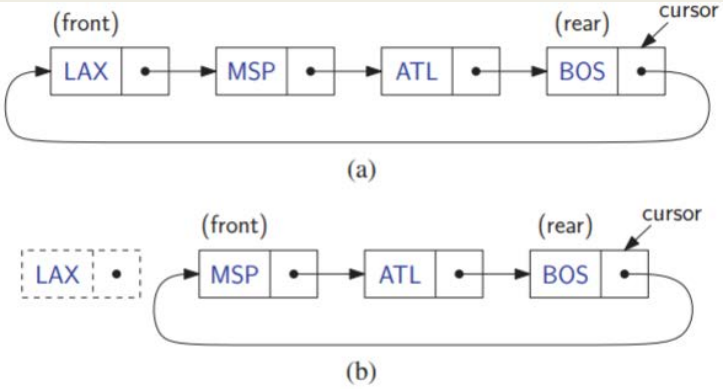

Figure 5.6: Dequeueing an element (in this case "LAX") from the front queue represented as a circularly linked list: (a) before the operation; (b) after removing the node immediately following the cursor.

```
typedef string Elem;                          // queue element type
class LinkedQueue {                           // queue as doubly linked list
public:
  LinkedQueue();                              // constructor
  int size() const;                           // number of items in the queue
  bool empty() const;                         // is the queue empty?
  const Elem& front() const throw(QueueEmpty); // the front element
  void enqueue(const Elem& e);                // enqueue element at rear
  void dequeue() throw(QueueEmpty);           // dequeue element at front
private:                                      // member data
  CircleList C;                               // circular list of elements
  int n;                                      // number of elements
};
```

Code Fragment 5.18: The class LinkedQueue, an implementation of a queue based on a circularly linked list.

```
                                              // enqueue element at rear
void LinkedQueue::enqueue(const Elem& e) {
  C.add(e);                                   // insert after cursor
  C.advance();                                // ...and advance
  n++;
}
                                              // dequeue element at front
void LinkedQueue::dequeue() throw(QueueEmpty) {
  if (empty())
    throw QueueEmpty("dequeue of empty queue");
  C.remove();                                 // remove from list front
  n--;
}
```

Code Fragment 5.20: The enqueue and dequeue functions for LinkedQueue.

Double-Ended Queues (§ 5.3)

- Double-Ended Queues (Deque pronounced “deck”)
  - *Member functions (STL has different names)*
    - insertFront(e)
    - insertBack(e)
    - eraseFront()
    - eraseBack()
    - front()
    - back()
    - size()
    - empty()
  - *Every operation is O*(1).
- Note, STL deque manages its elements with a dynamic array and thus provides random access!

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()	–	()

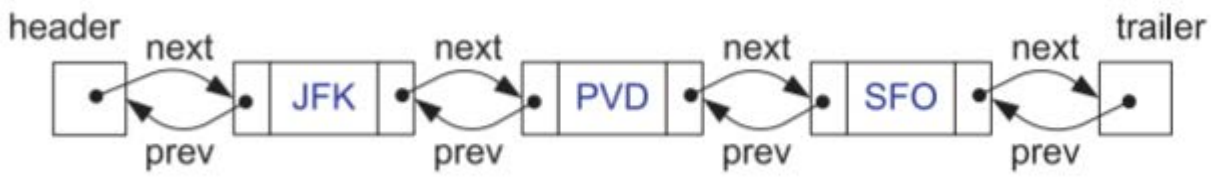

Figure 5.7: A doubly linked list with sentinels, *header* and *trailer*. The front of our deque is stored just after the header ("JFK"), and the back of our deque is stored just before the trailer ("SFO").

```
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
};
```

Code Fragment 5.21: The class structure for class LinkedDeque.

```
                                                    // 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--;
}
```

Code Fragment 5.22: The insertion and removal functions for LinkedDeque.

```
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
};
```

Code Fragment 5.23: Implementation of the Stack interface by means of a deque.

```
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();
}
```

Code Fragment 5.24: Implementation of the Stack interface by means of a deque.