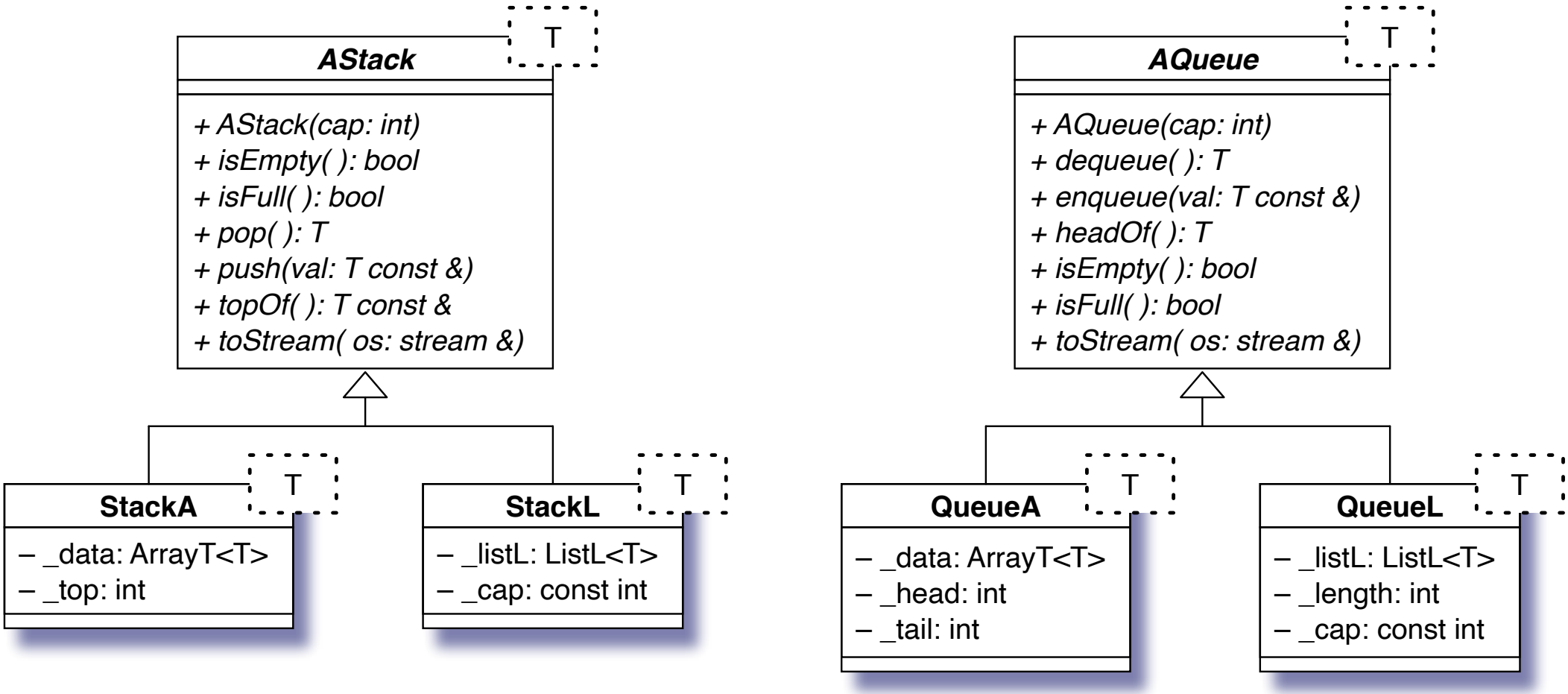
Stacks and Queues

Figure 7.1


AStack
T
+ AStack(cap: int)
+ isEmpty( ): bool
+ isFull( ): bool
+ pop( ): T
+ push(val: T const &)
+ topOf( ): T const &
+ toStream(os: stream &)
StackA
T
– _data: ArrayT<T>
– _top: int
StackL
T
– _listL: ListL<T>
– _cap: const int
AQueue
T
+ AQueue(cap: int)
+ dequeue( ): T
+ enqueue(val: T const &)
+ headOf( ): T
+ isEmpty( ): bool
+ isFull( ): bool
+ toStream(os: stream &)
QueueA
T
– _data: ArrayT<T>
– _head: int
– _tail: int
QueueL
T
– _listL: ListL<T>
– _length: int
– _cap: const int

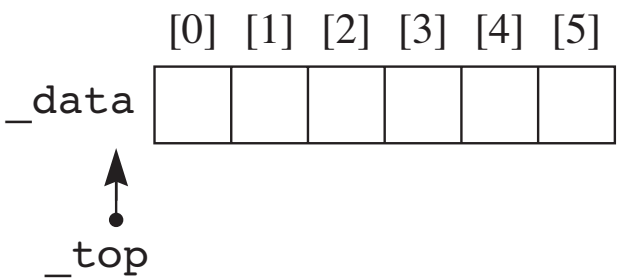
Demo StackA

An array implementation of a stack

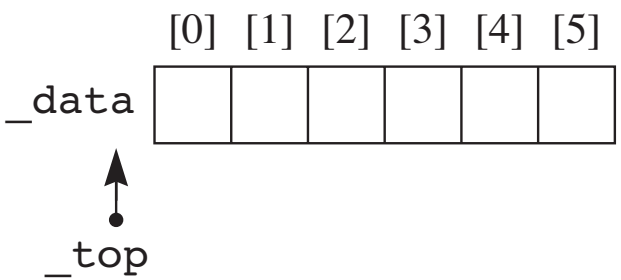
- A safe array of data elements: `_data`
- An integer pointer to the top of the stack: `_top`

```
// ========= StackA =========
template<class T>
class StackA : public AStack<T> {
private:
    ArrayT<T> _data;
    int _top;

public:
    explicit StackA(int cap = 1);
    // Post: This stack is allocated with a capacity of cap
    // and initialized to be empty.

    bool isEmpty() const override;
    // Post: true is returned if this stack is empty;
    // otherwise, false is returned.

    bool isFull() const override;
    // Post: true is returned if this stack is full;
    // otherwise, false is returned.
```

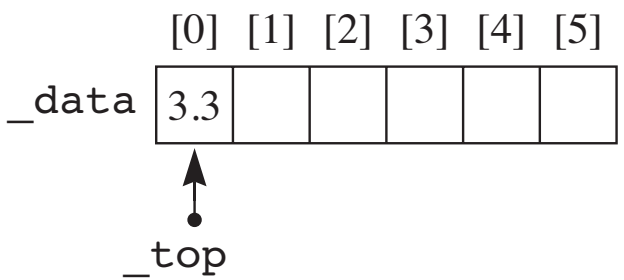
```
    T pop() override;
    // Pre: This stack is not empty.
    // Post: The top value in this stack is removed and returned.

    void push(T const &val) override;
    // Pre: This stack is not full.
    // Post: val is stored on top of this stack.

    T const &topOf() const override;
    // Pre: This stack is not empty.
    // Post: The top value from this stack is returned.

    void toStream(ostream &os) const override;
    // Post: All the items on this stack from top to bottom
    // are written to os.
};
```



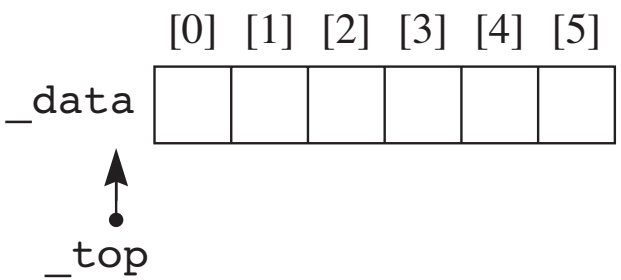
(a) Initial state.



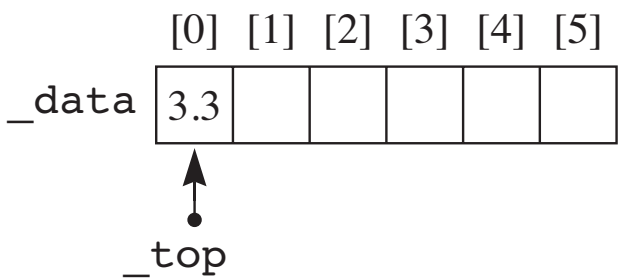
(a) Initial state.



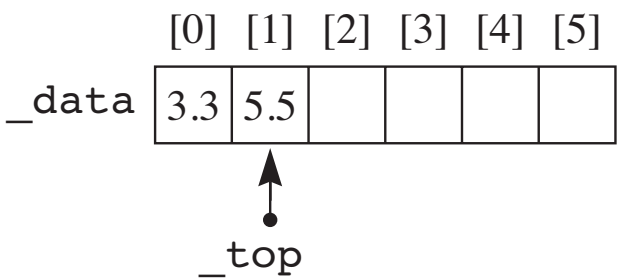
(b) `push(3.3);`



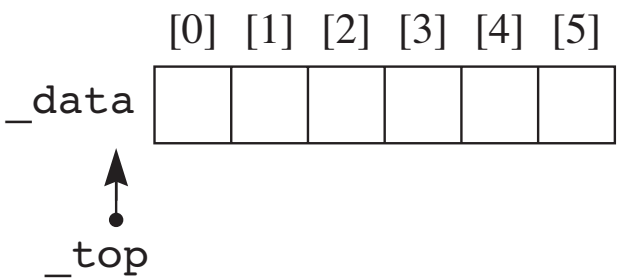
(a) Initial state.



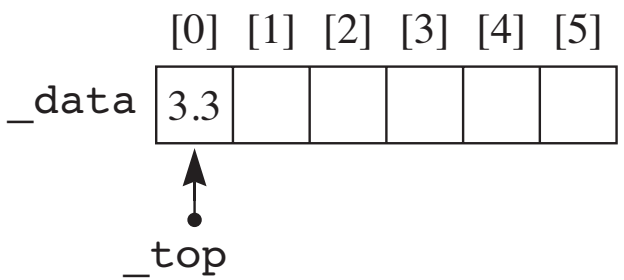
(b) `push(3.3);`



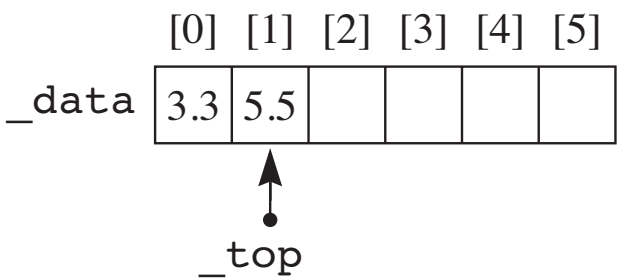
(c) `push(5.5);`



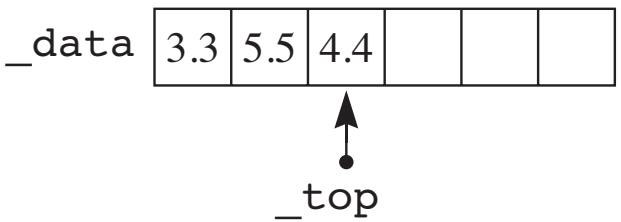
(a) Initial state.



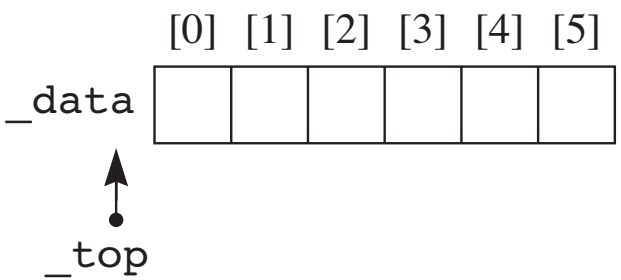
(b) `push(3.3);`



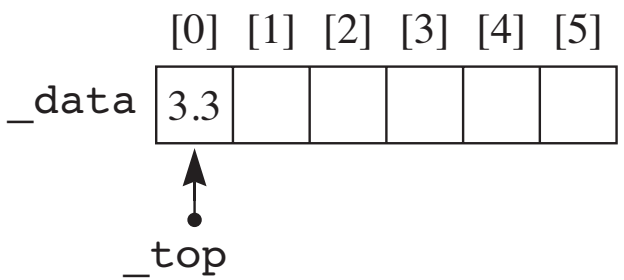
(c) `push(5.5);`



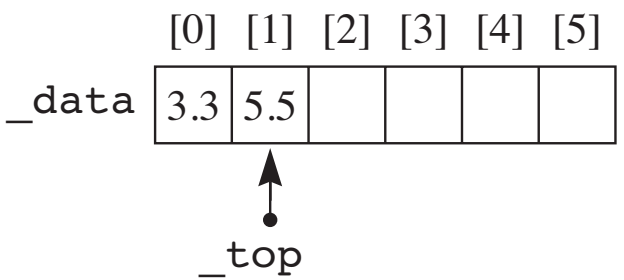
(d) `push(4.4);`



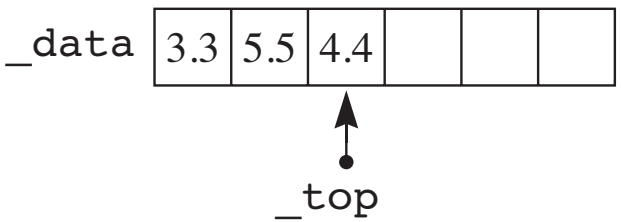
(a) Initial state.



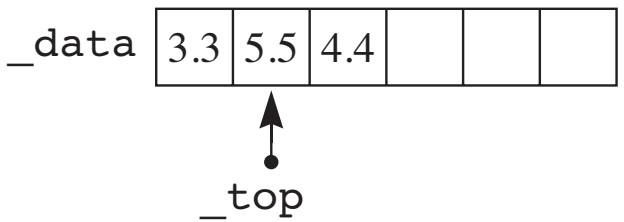
(b) `push(3.3);`



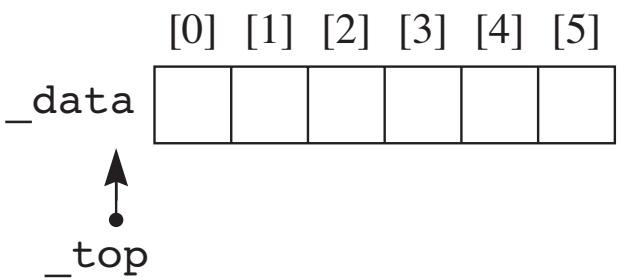
(c) `push(5.5);`



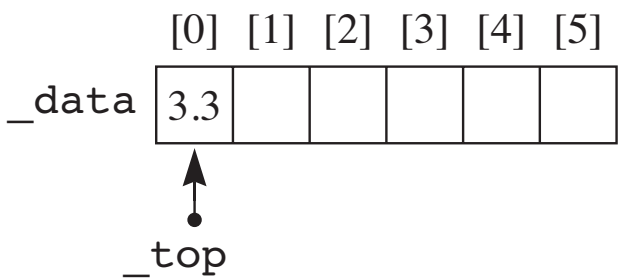
(d) `push(4.4);`



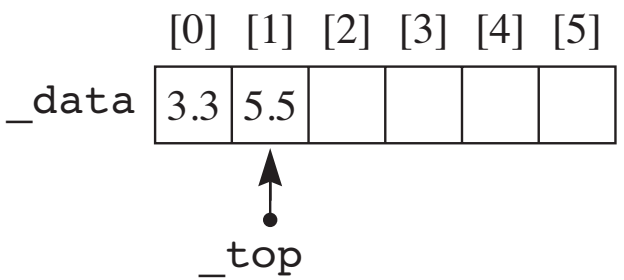
(d) `pop();` Returns 4.4.



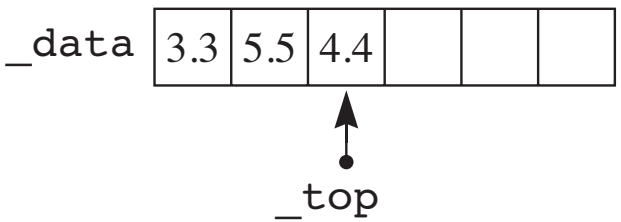
(a) Initial state.



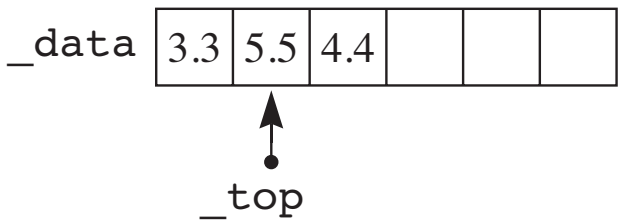
(b) `push(3.3);`



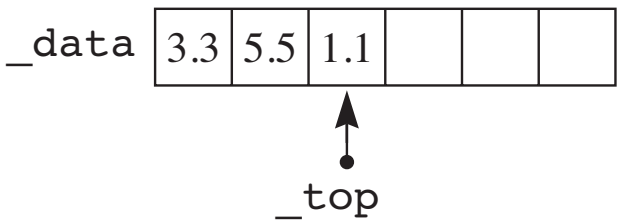
(c) `push(5.5);`



(d) `push(4.4);`



(d) `pop();` Returns 4.4.



(e) `push(1.1);`

```
// ========= Constructor =========
template<class T>
StackA<T>::StackA(int cap):
    _data(cap),
    _top(-1) {
}

// ========= isEmpty =========
template<class T>
bool StackA<T>::isEmpty() const {
    cerr << "isEmpty: Exercise for the student." << endl;
    throw -1;
}

// ========= isFull =========
template<class T>
bool StackA<T>::isFull() const {
    cerr << "isFull: Exercise for the student." << endl;
    throw -1;
}
```

```
// ========= pop =========
template<class T>
T StackA<T>::pop() {
    if (isEmpty()) {
        cerr << "pop precondition violated: "
             << "Cannot pop from an empty stack." << endl;
        throw -1;
    }
    cerr << "pop: Exercise for the student." << endl;
    throw -1;
}

// ========= push =========
template<class T>
void StackA<T>::push(T const &val) {
    cerr << "push: Exercise for the student." << endl;
    throw -1;
}
```

```
// ========= topOf =========
template<class T>
T const &StackA<T>::topOf() const {
    if (isEmpty()) {
        cerr << "topOf precondition violated: "
             << "An empty stack has no top." << endl;
        throw -1;
    }
    cerr << "topOf: Exercise for the student." << endl;
    throw -1;
}
```

```
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, StackA<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void StackA<T>::toStream(ostream &os) const {
    os << "(";
    for (int i = _top; i > 0; i--) {
        os << _data[i] << ", ";
    }
    if (_top == -1) {
        os << ")";
    }
    else {
        os << _data[0] << ")";
    }
}
```

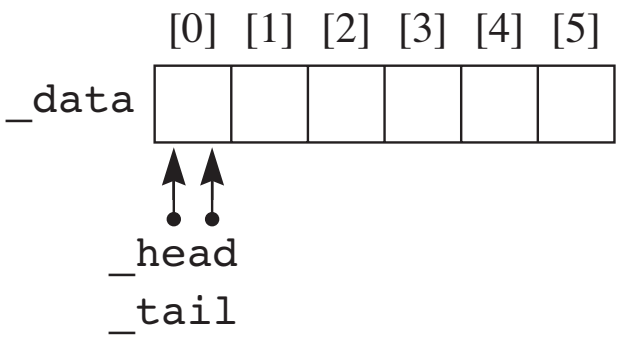
Demo QueueA

An array implementation of a queue

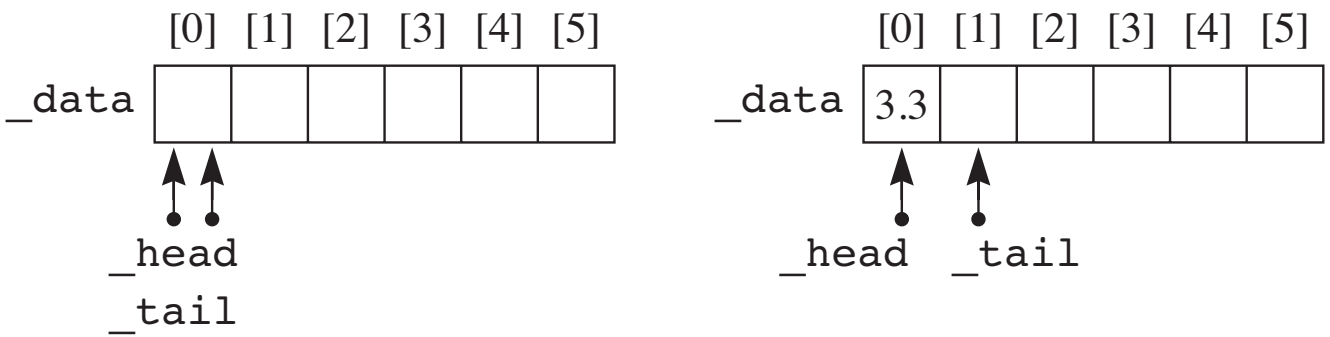
- A safe array of data elements: `_data`
- An integer pointer to the head of the queue: `_head`
- An integer pointer to the tail of the queue: `_tail`
- The queue is circular

```
// ========= QueueA =========
template<class T>
class QueueA : public AQueue<T> {
private:
    ArrayT<T> _data;
    int _head, _tail;

public:
    explicit QueueA(int cap = 1);
    // Post: This queue is allocated with a capacity of cap
    // and initialized to be empty.

    T dequeue() override;
    // Pre: This queue is not empty.
    // Post: The head value in this queue is removed and returned.

    void enqueue(T const &val) override;
    // Pre: This queue is not full.
    // Post: val is stored at the tail of this queue.
```

```
    T const &headOf() const override;
    // Pre: This queue is not empty.
    // Post: The head value from this queue is returned.

    bool isEmpty() const override;
    // Post: true is returned if this queue is empty;
    // otherwise, false is returned.

    bool isFull() const override;
    // Post: true is returned if this queue is full;
    // otherwise, false is returned.

    void toStream(ostream &os) const override;
    // Post: All the items on this queue from tail to head
    // are written to os.
};
```

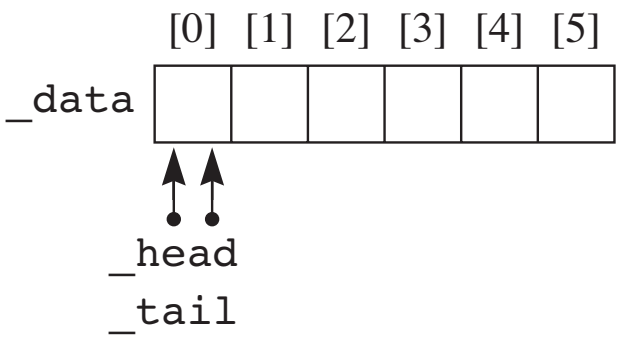


(a) Initial state.

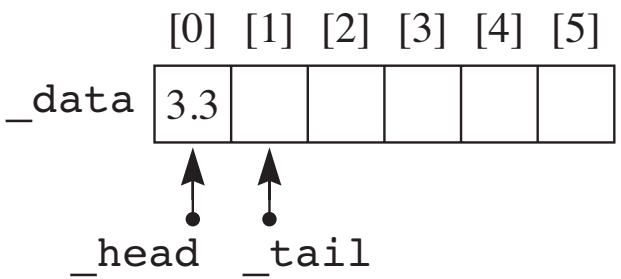


(a) Initial state.

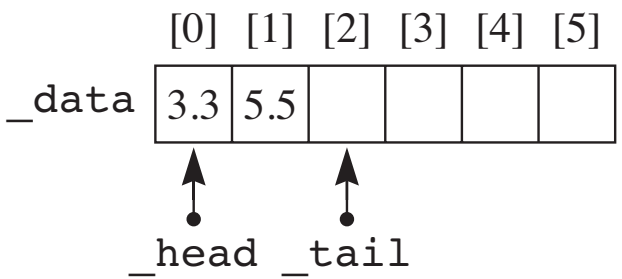
(b) `enqueue(3.3);`



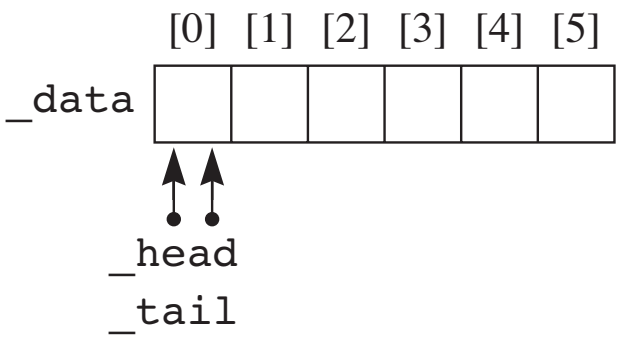
(a) Initial state.



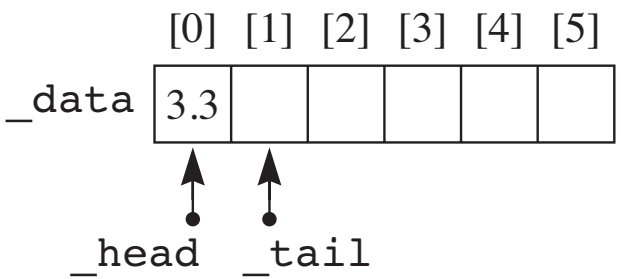
(b) `enqueue(3.3);`



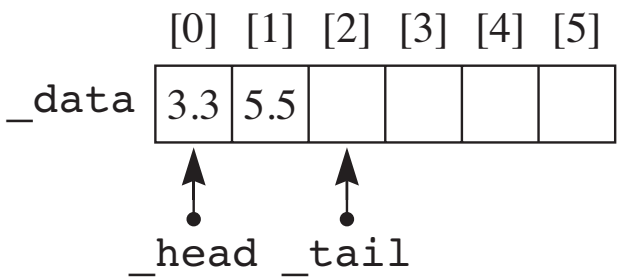
(c) `enqueue(5.5);`



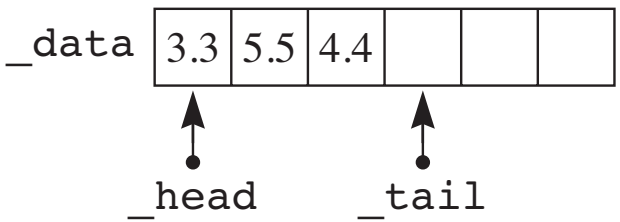
(a) Initial state.



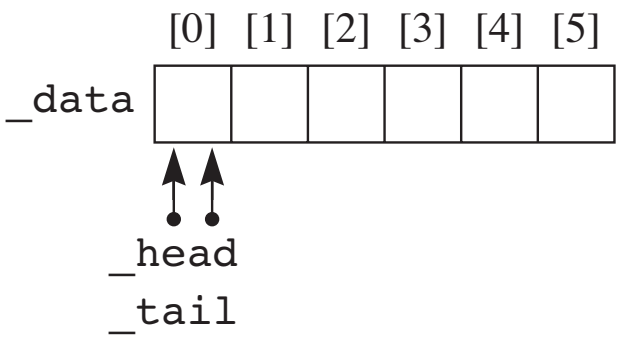
(b) `enqueue(3.3);`



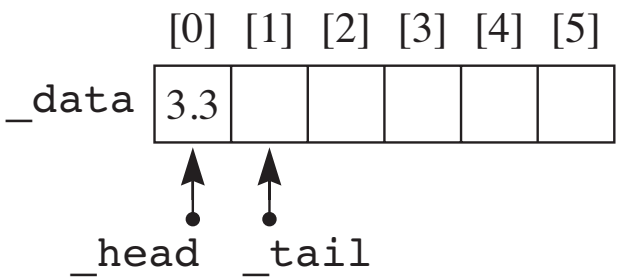
(c) `enqueue(5.5);`



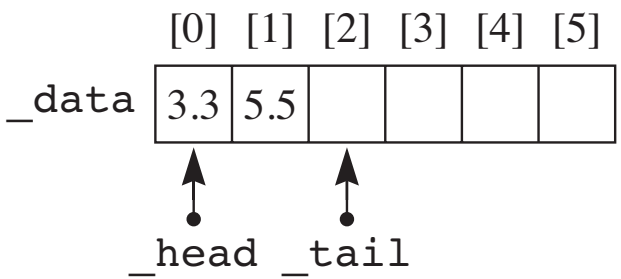
(d) `enqueue(4.4);`



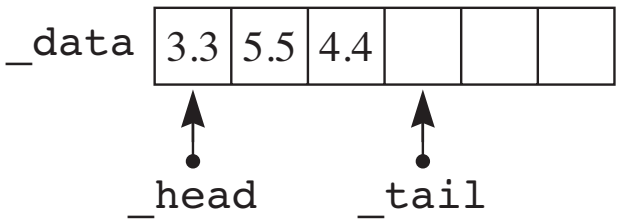
(a) Initial state.



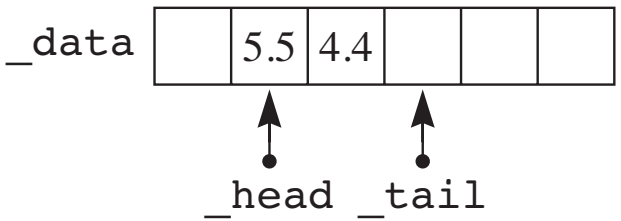
(b) `enqueue(3.3);`



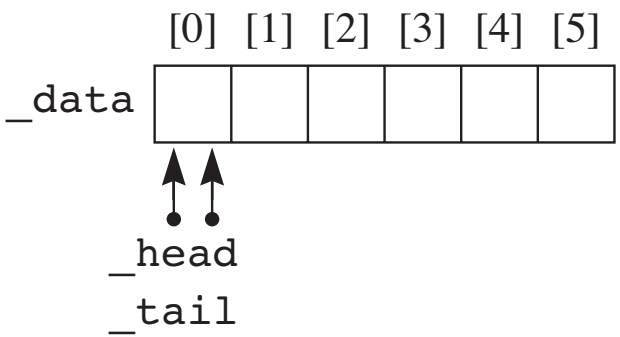
(c) `enqueue(5.5);`



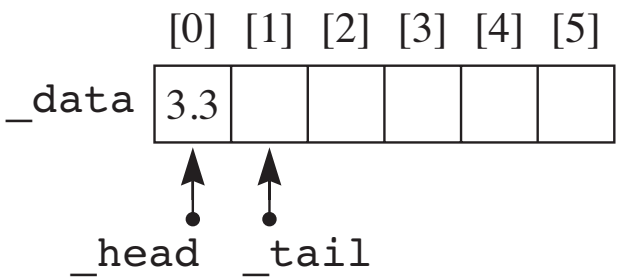
(d) `enqueue(4.4);`



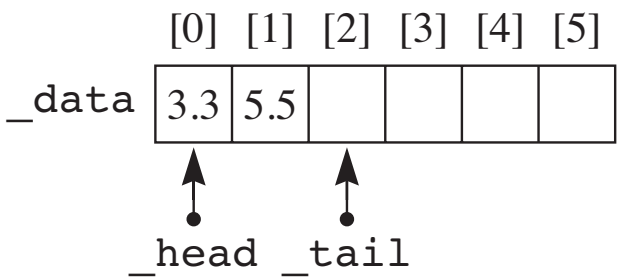
(e) `dequeue();`
Returns 3.3.



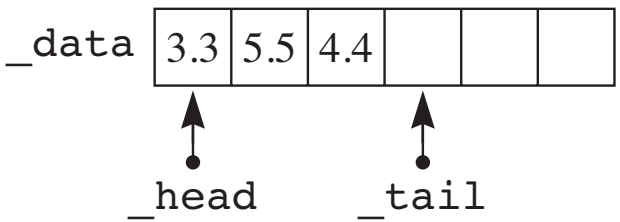
(a) Initial state.



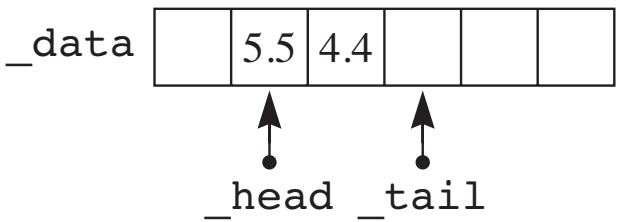
(b) `enqueue(3.3);`



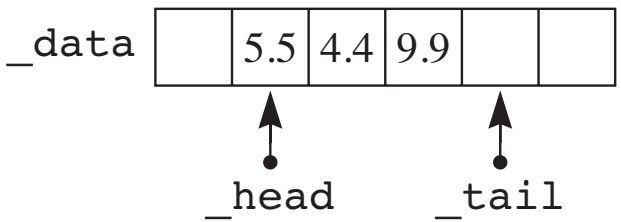
(c) `enqueue(5.5);`



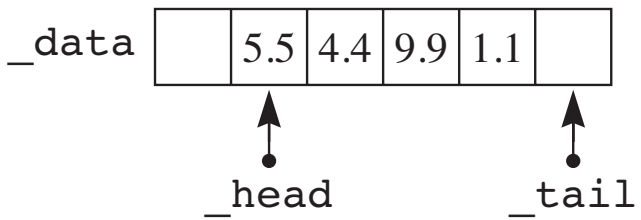
(d) `enqueue(4.4);`



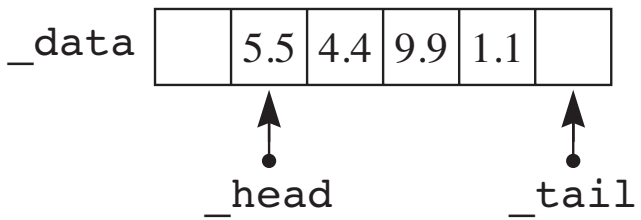
(e) `dequeue();`
Returns 3.3.



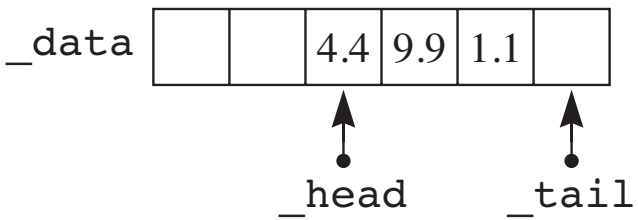
(f) `enqueue(9.9);`



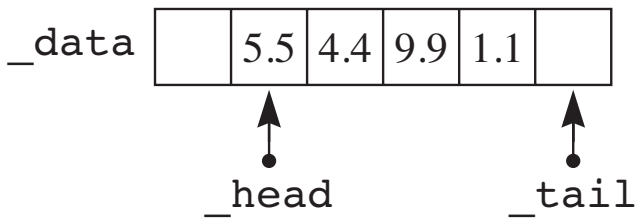
(g) `enqueue(1.1);`



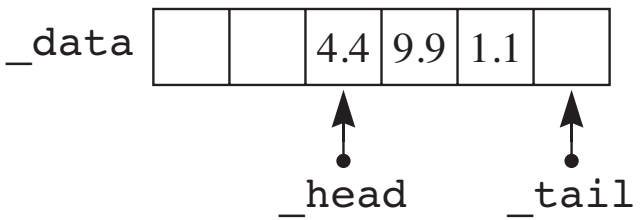
(g) `enqueue(1.1);`



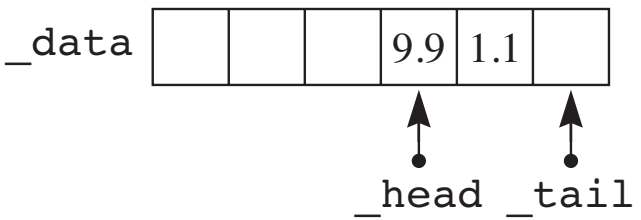
(h) `dequeue();`
Returns 5.5.



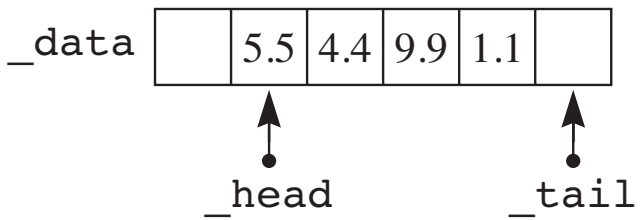
(g) `enqueue(1.1);`



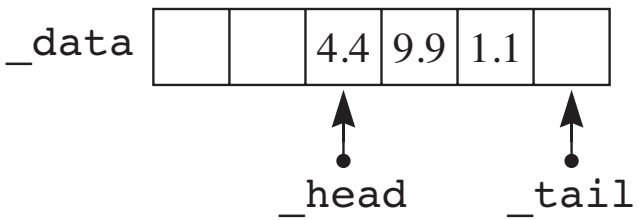
(h) `dequeue();`
Returns 5.5.



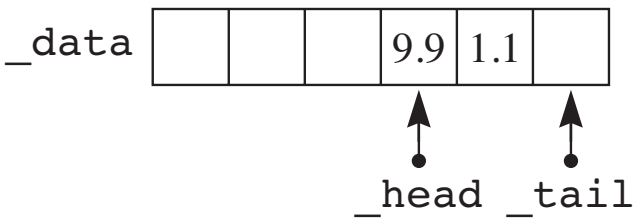
(i) `dequeue();`
Returns 4.4.



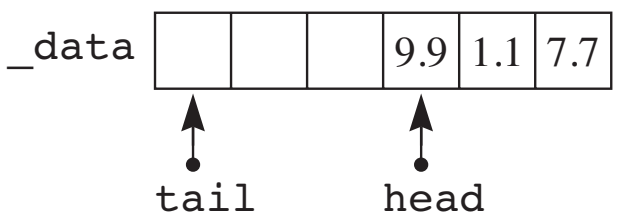
(g) `enqueue(1.1);`



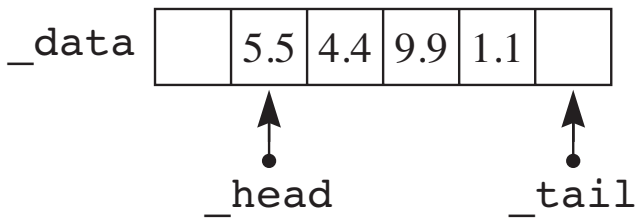
(h) `dequeue();`
Returns 5.5.



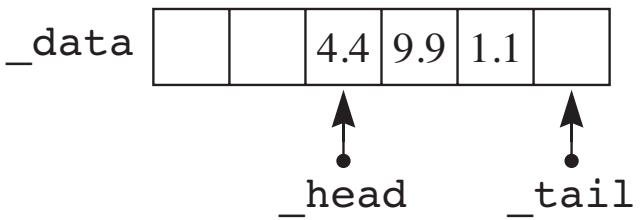
(i) `dequeue();`
Returns 4.4.



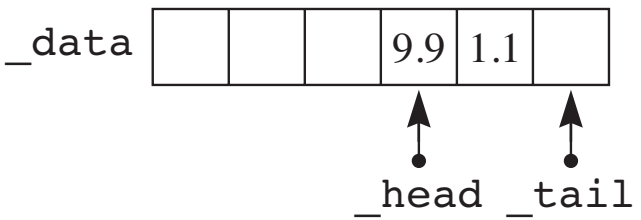
(j) `enqueue(7.7);`



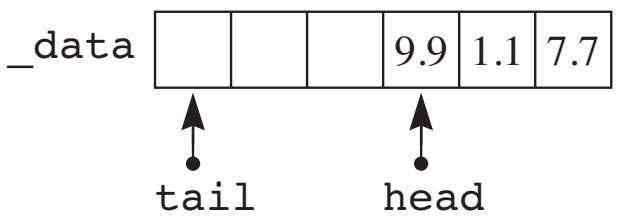
(g) `enqueue(1.1);`



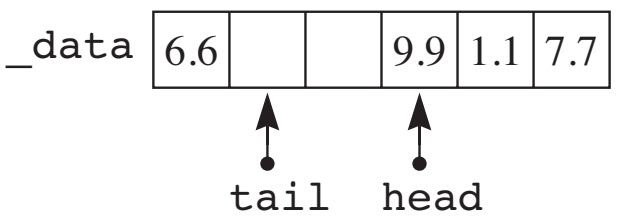
(h) `dequeue();`
Returns 5.5.



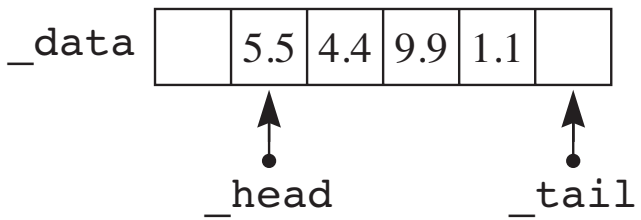
(i) `dequeue();`
Returns 4.4.



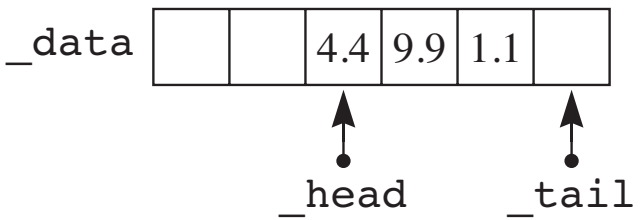
(j) `enqueue(7.7);`



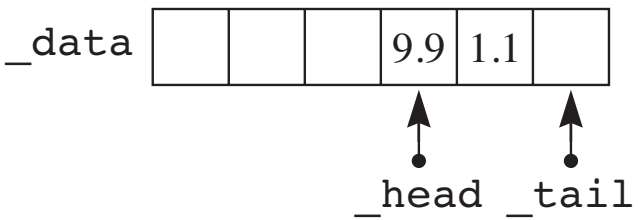
(k) `enqueue(6.6);`



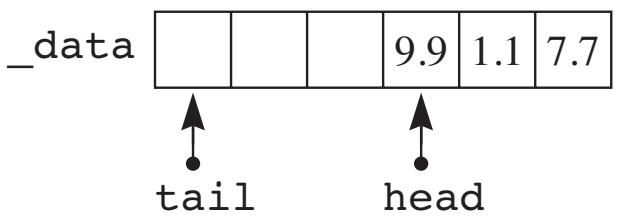
(g) `enqueue(1.1);`



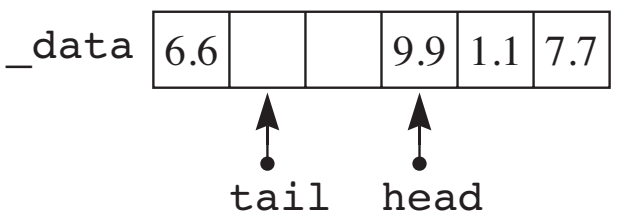
(h) `dequeue();`
Returns 5.5.



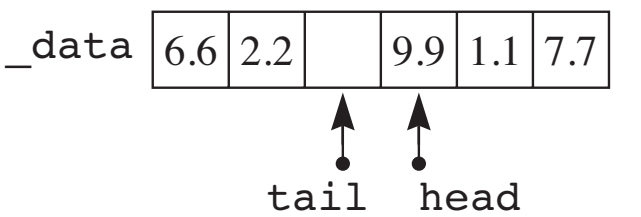
(i) `dequeue();`
Returns 4.4.



(j) `enqueue(7.7);`



(k) `enqueue(6.6);`



(l) `enqueue(2.2);`
The queue is full.

```
// ========= Constructor =========
template<class T>
QueueA<T>::QueueA(int cap) :
    _data(cap + 1),
    _head(0),
    _tail(0) {
}

// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, QueueA<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void QueueA<T>::toStream(ostream &os) const {
    cerr << "toStream: Exercise for the student." << endl;
    throw -1;
}
```

The Adapter Design Pattern
Also known as the Wrapper Pattern

Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.

A linked list implementation of the stack

The stack wraps its own interface around that of the already existing linked list.

```
// ========= StackL =========
template<class T>
class StackL : public AStack<T> {
private:
    ListL<T> _listL;
    const int _cap;

public:
    StackL(int cap = 1);
    // This stack is initialized to be empty.

    bool isEmpty() const override;
    // Post: true is returned if this stack contains no element;
    // otherwise, false is returned.

    bool isFull() const override;
    // Post: true is returned if the number of elements in this stack
    // is equal to its cap;
    // otherwise, false is returned.
```

```
    T pop() override;
    // Pre: This stack is not empty.
    // Post: The top value in this stack is removed and returned.

    void push(T const &val) override;
    // Post: val is stored on top of this stack.

    T const &topOf() const override;
    // Pre: This stack is not empty.
    // Post: The top value from this stack is returned.

    void toStream(ostream &os) const override;
    // Post: All the items on this stack from top to bottom
    // are written to os.
};
```

```
// ========= Constructor =========
template<class T>
StackL<T>::StackL(int cap):
    _listL(),
    _cap(cap) {
}

// ========= isEmpty =========
template<class T>
bool StackL<T>::isEmpty() const {
    return _listL.isEmpty();
}

// ========= isFull =========
template<class T>
bool StackL<T>::isFull() const {
    return _listL.length() == _cap;
}
```

```
// ========== push ==========
template<class T>
void StackL<T>::push(const T &val) {
    if (isFull()) {
        cerr << "push precondition violated: "
             << "Cannot push onto a full stack." << endl;
        throw -1;
    }
    _listL.prepend(val);
}
```

```
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, StackL<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void StackL<T>::toStream(ostream &os) const {
    _listL.toStream(os);
}
```

A linked list implementation of the queue

The queue wraps its own interface around that of the already existing linked list.

```
// ========= QueueL =========
template<class T>
class QueueL : public AQueue<T> {
private:
    // Attribute is exercise for the student.

public:
    QueueL(int cap = 1);
    // This queue is allocated and initialized to be empty.

    T dequeue() override;
    // Pre: This queue is not empty.
    // Post: The head value in this queue is removed and returned.

    void enqueue(T const &val) override;
    // Post: val is stored at the tail of this queue.
```

```
    T const &headOf() const override;
    // Pre: This queue is not empty.
    // Post: The head value from this queue is returned.

    bool isEmpty() const override;
    // Post: true is returned if this queue contains no element;
    // otherwise, false is returned.

    bool isFull() const override;
    // Post: true if the number of elements contained
    // in this queue is equal to the queue's capacity;
    // otherwise false is returned.

    void toStream(ostream &os) const override;
    // Post: All the items on this queue from tail to head
    // are written to os.
};
```

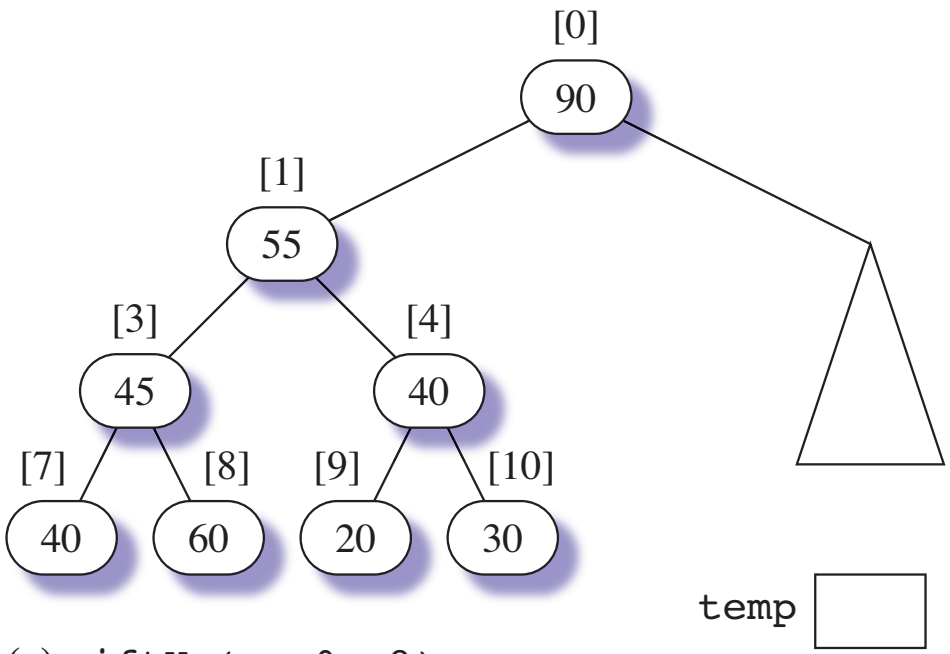
The Priority Queue

A heap implementation of the priority queue

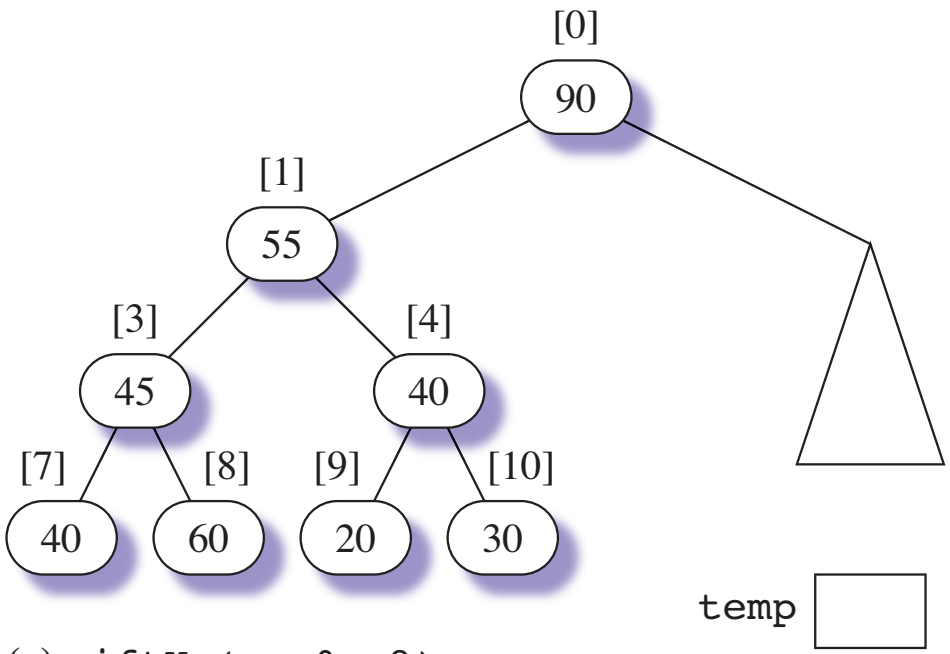
```
// ========= PriorityQ =========
template<class T>
class PriorityQ {
private:
    ArrayT<T> _data;
    int _hiIndex;

public:
    PriorityQ(int cap);
    // Post: This priority queue is allocated with a capacity of cap
    // and initialized to be empty.

    T extractMax();
    // Pre: This priority queue is not empty.
    // Post: The maximum value in this priority queue is removed
    // and returned.
```

```
int heapSize() const;
// Post: The size of this priority queue is returned.

void increaseKey(int i, T const &key);
// Pre: This priority queue is not empty, 0 <= i < heapSize(),
// and key is at least as large as the key at index i.
// Post: The value of the element at index i is increased to key.

void insert(T const &val);
// Pre: This priority queue is not full.
// Post: val is stored in this priority queue.
```

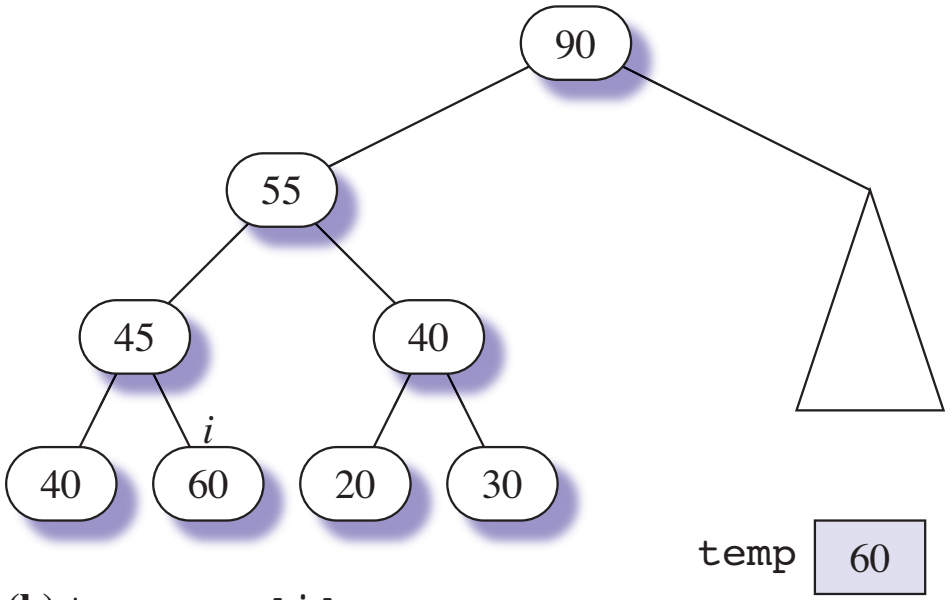
```
    bool isEmpty() const;
    // Post: true is returned if this priority queue is full;
    // otherwise, false is returned.

    bool isFull() const;
    // Post: true is returned if this priority queue is full;
    // otherwise, false is returned.

    T const &maximum() const;
    // Pre: This priority queue is not empty.
    // Post: The maximum value from this priority queue is returned.

    void toStream(ostream &os) const;
    // Post: Each item on this priority queue prefixed with its index
    // is written to os.
};
```



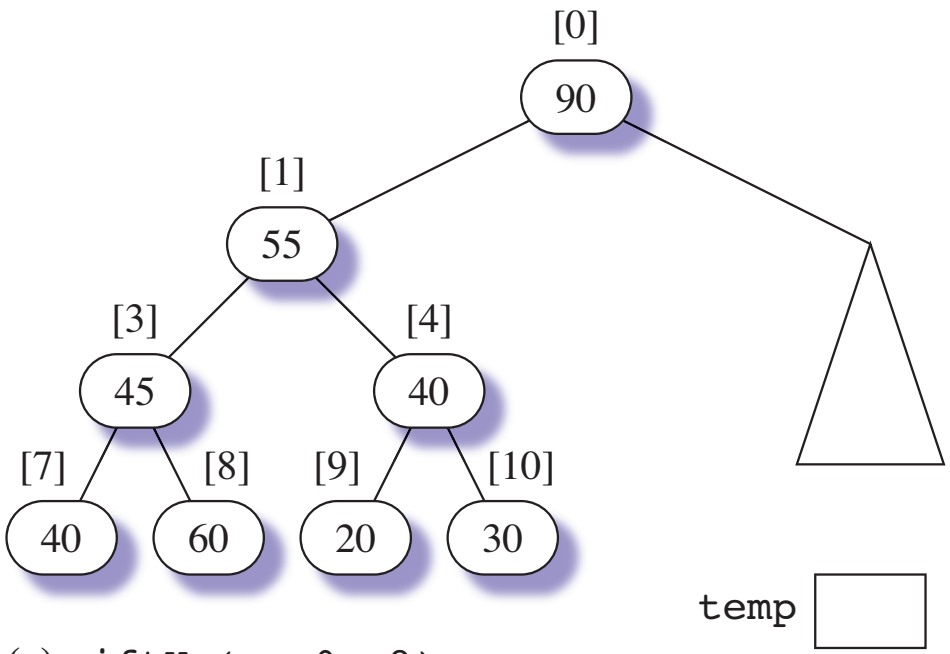
(a) `siftUp(a, 0, 8);`



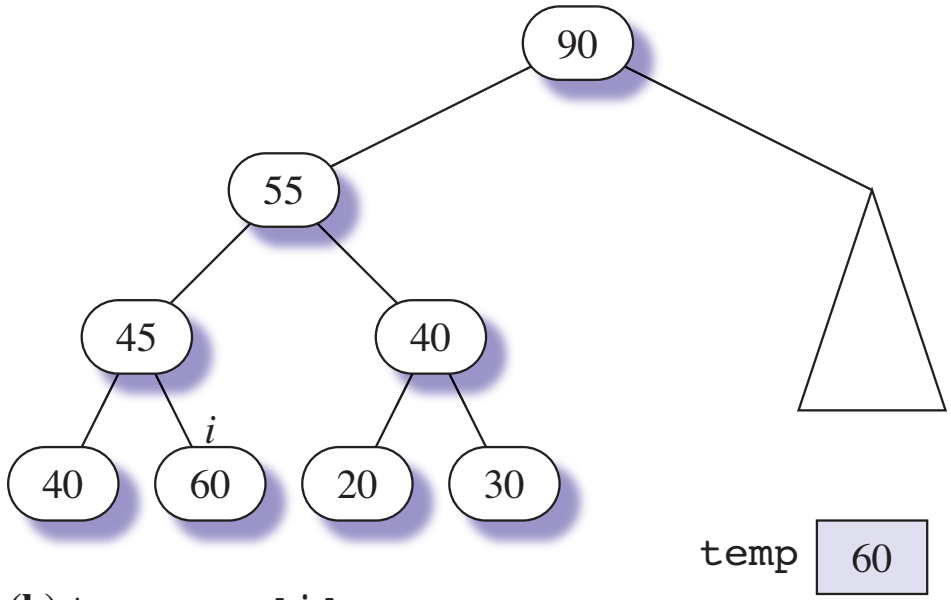
(a) `siftUp(a, 0, 8);`



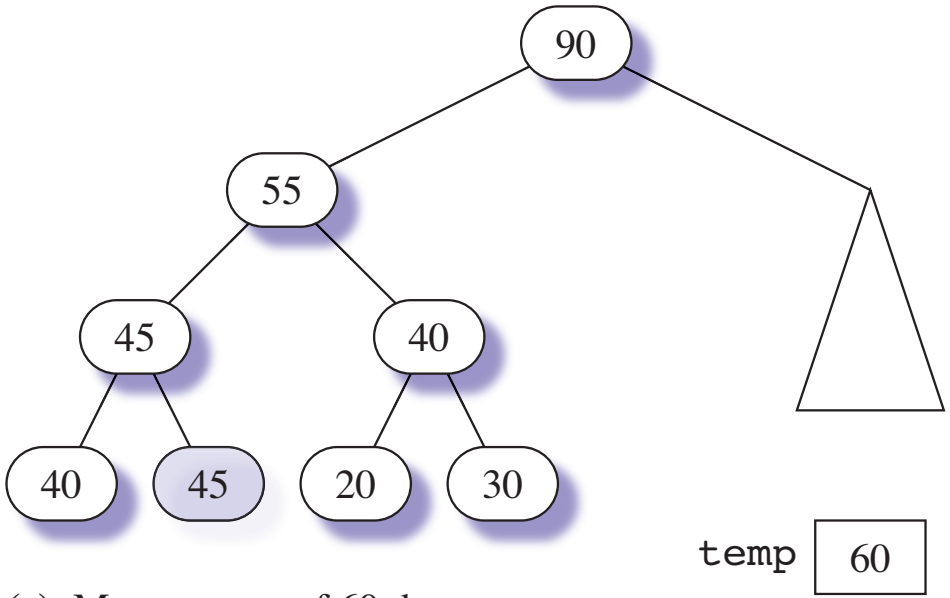
(b) `temp = a[i];`



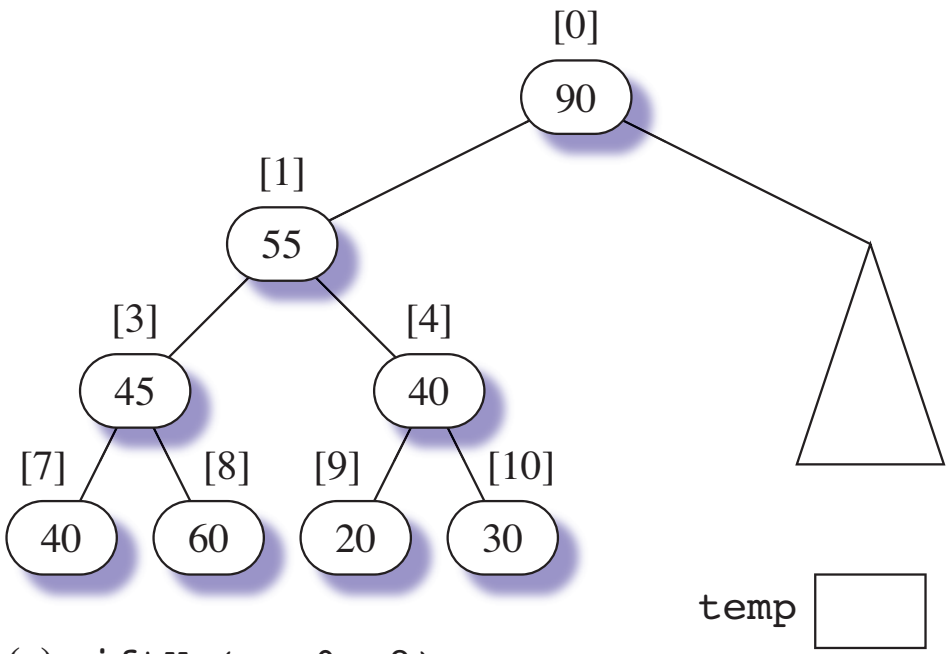
(a) `siftUp(a, 0, 8);`



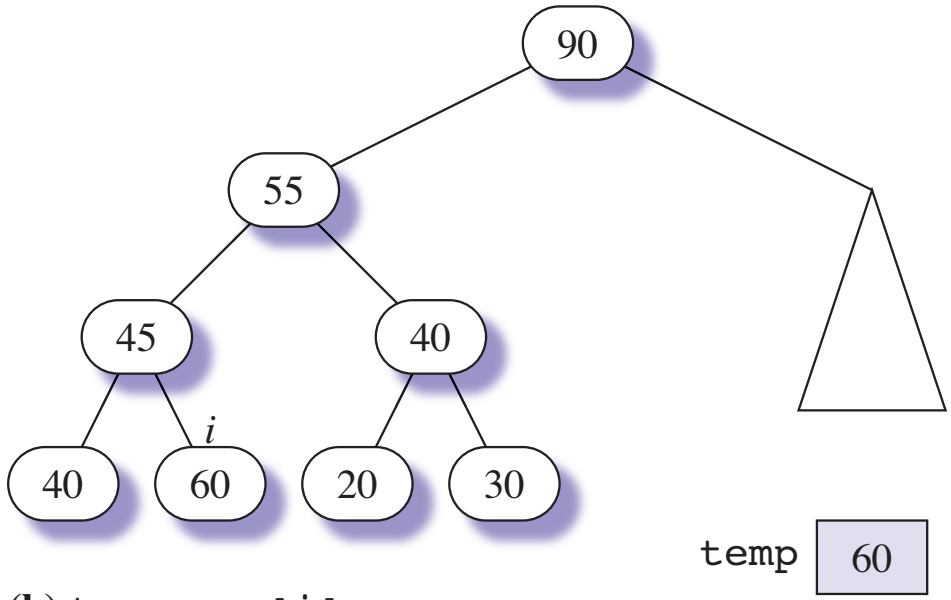
(b) `temp = a[i];`



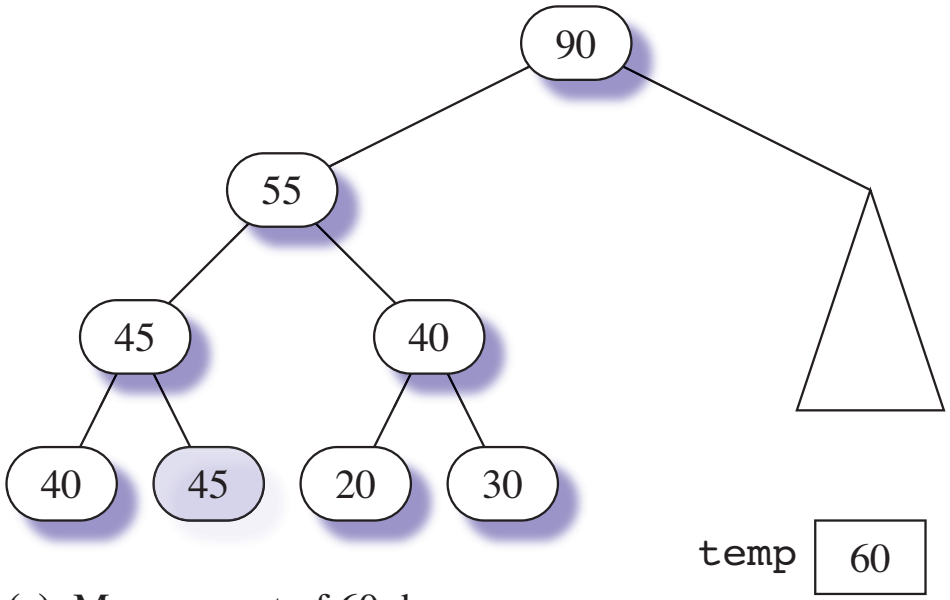
(c) Move parent of 60 down.



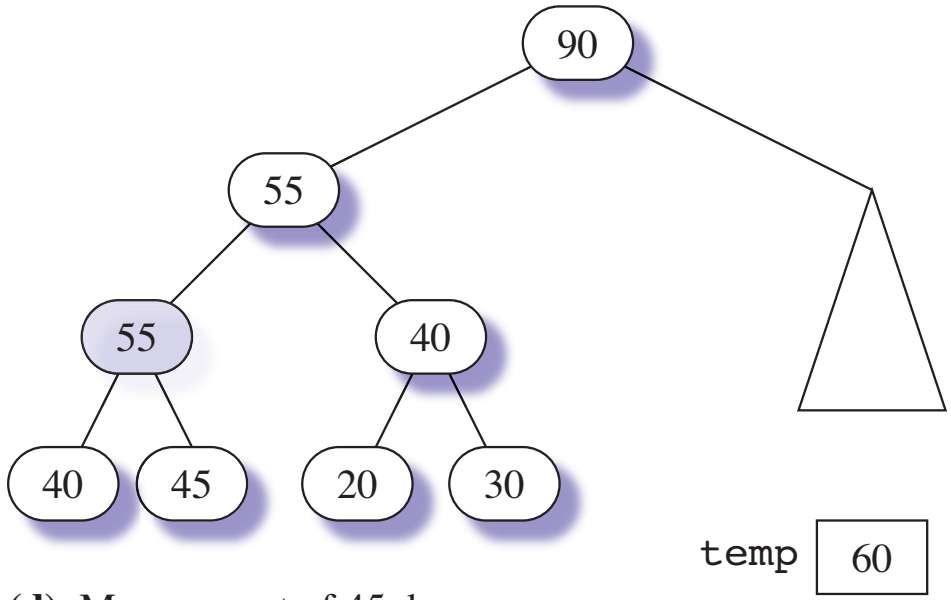
(a) `siftUp(a, 0, 8);`



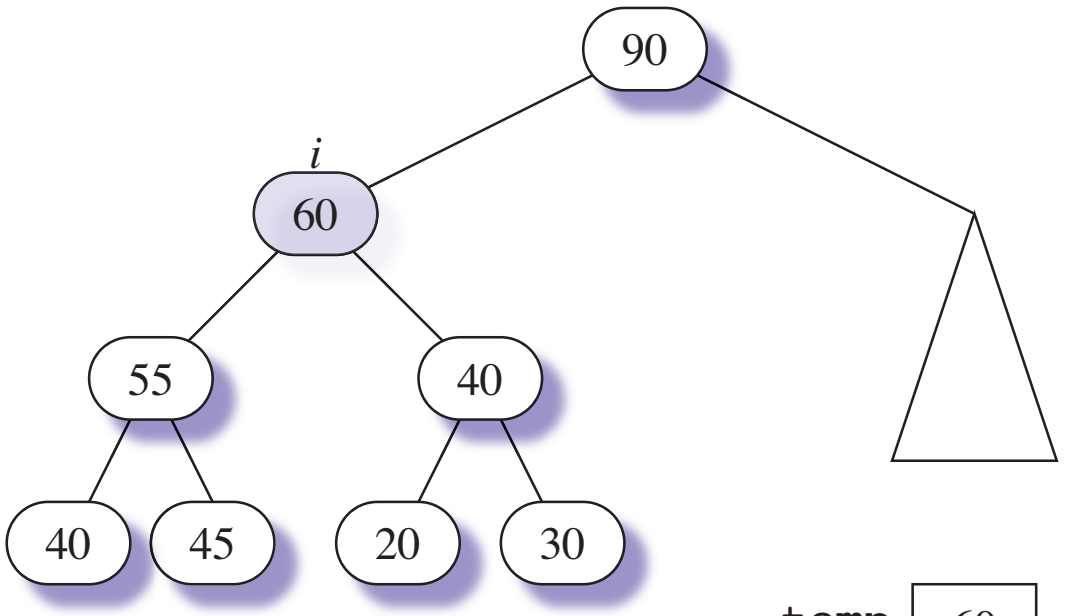
(b) `temp = a[i];`



(c) Move parent of 60 down.



(d) Move parent of 45 down.



(e) `a[i] = temp;`

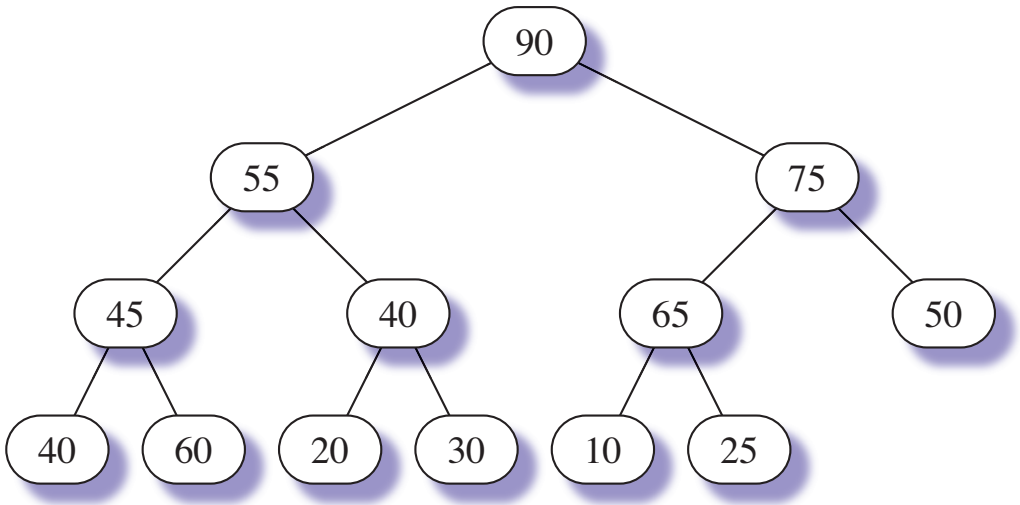
```
// ========= siftUp =========
template<class T>
void siftUp(ASeq<T> &a, int lo, int i) {
    // Pre: maxHeap(a[lo..i - 1]).
    // Post: maxHeap(a[lo..i]).
    T temp = a[i];
    int parent = (i + lo - 1) / 2;
    while (lo < i && a[parent] < temp) {
        cerr << "siftUp: Exercise for the student." << endl;
        throw -1;
    }
    a[i] = temp;
}
```

```
// ========= Constructor =========
template<class T>
PriorityQ<T>::PriorityQ(int cap) :
    _data(cap),
    _hiIndex(-1) {
}

// ========= heapSize =========
template<class T>
int PriorityQ<T>::heapSize() const {
    return _hiIndex + 1;
}
```

```
// ========= insert =========
template<class T>
void PriorityQ<T>::insert(T const &val) {
    if (isFull()) {
        cerr << "insert precondition violated: "
             << "Cannot insert into a full priority queue." << endl;
        throw -1;
    }
    _data[++_hiIndex] = val;
    siftUp(_data, 0, _hiIndex);
}
```

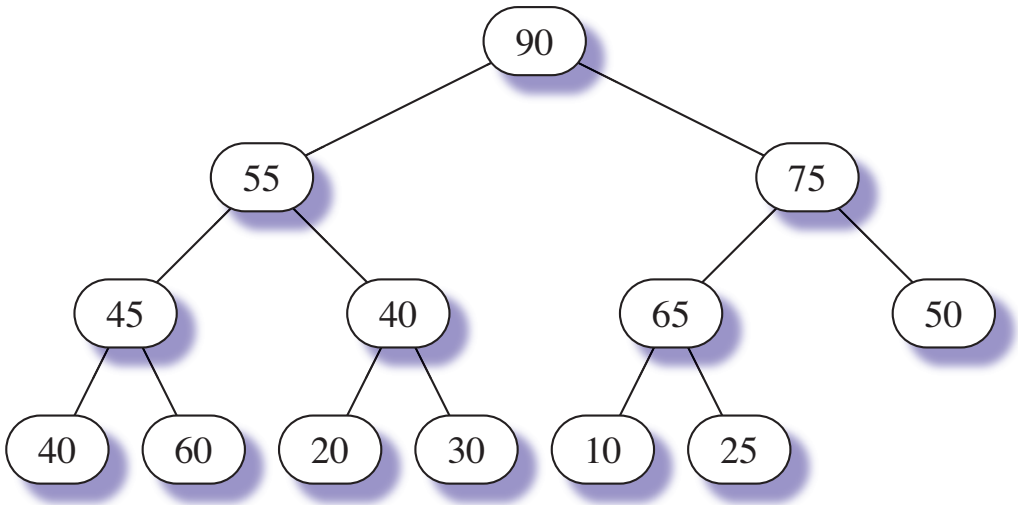
```
// ========= isEmpty =========
template<class T>
bool PriorityQ<T>::isEmpty() const {
    return _hiIndex == -1;
}

// ========= maximum =========
template<class T>
T const &PriorityQ<T>::maximum() const {
    if (isEmpty()) {
        cerr << "maximum precondition violated: "
             << "An empty priority queue has no maximum." << endl;
        throw -1;
    }
    return _data[0];
}
```

```
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, PriorityQ<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void PriorityQ<T>::toStream(ostream &os) const {
    for (int i = 0; i <= _hiIndex; i++) {
        os << i << ":" << _data[i] << "  ";
    }
}
```

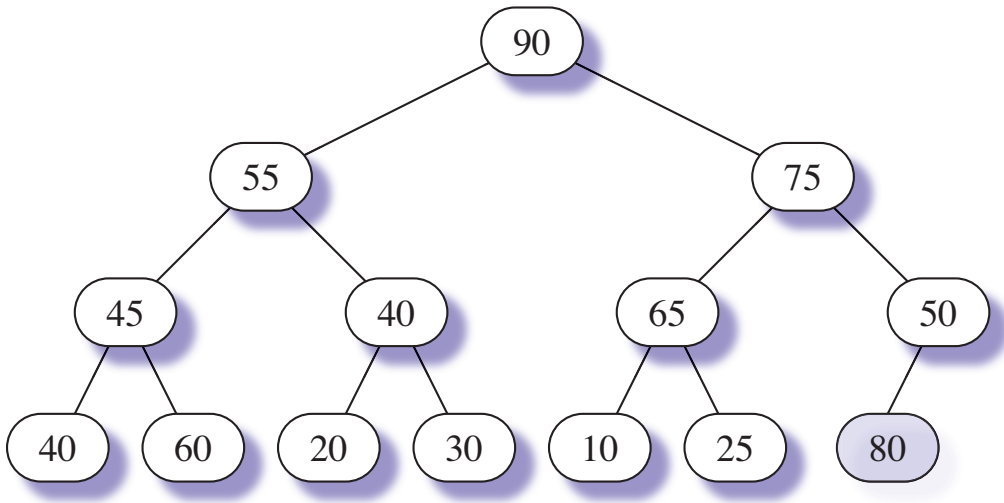


(a) `insert(80);`

Figure 7.17

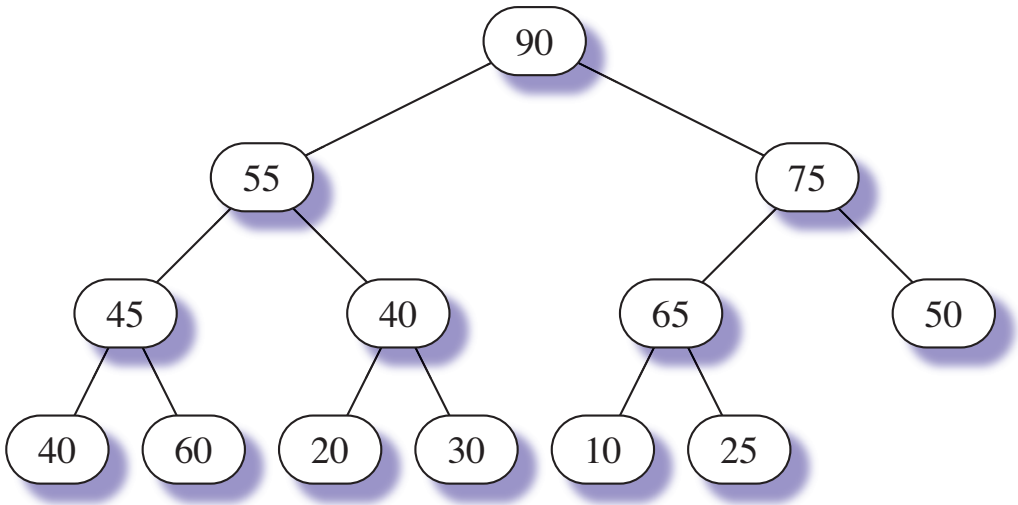


(a) `insert(80);`

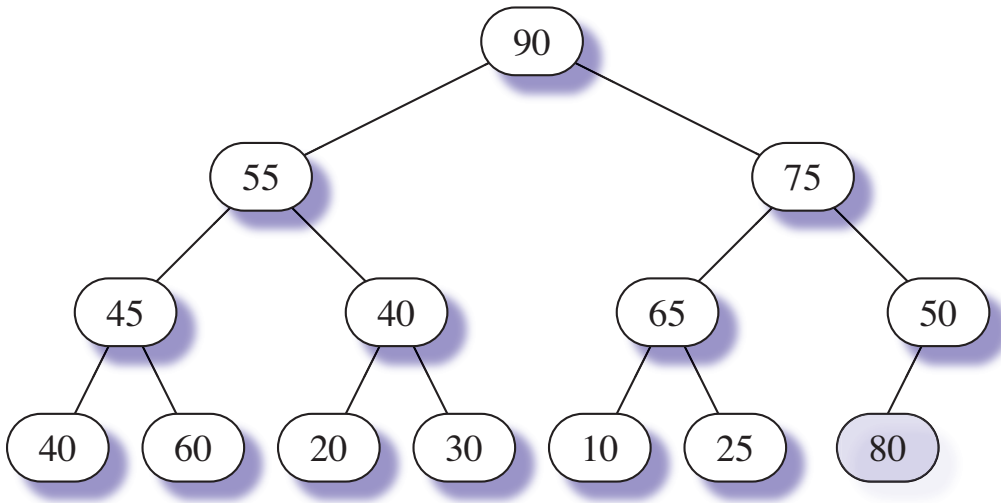


(b) `_data[++_hiIndex] = val;`

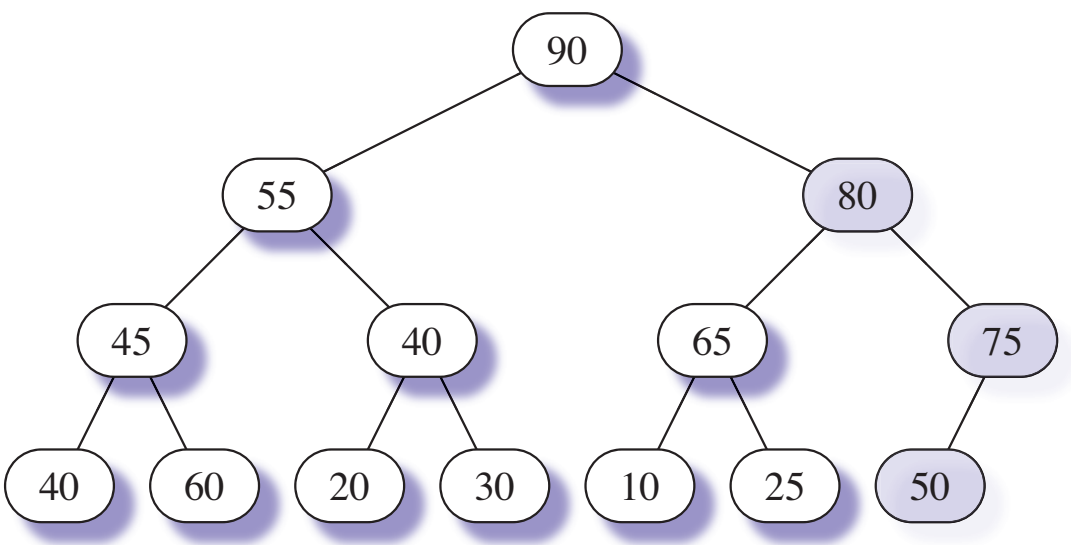
Figure 7.17



(a) `insert(80);`



(b) `_data[++_hiIndex] = val;`



(c) `siftUp(_data, 0, _hiIndex);`