The Template Method Pattern

Used when a system must implement an algorithm

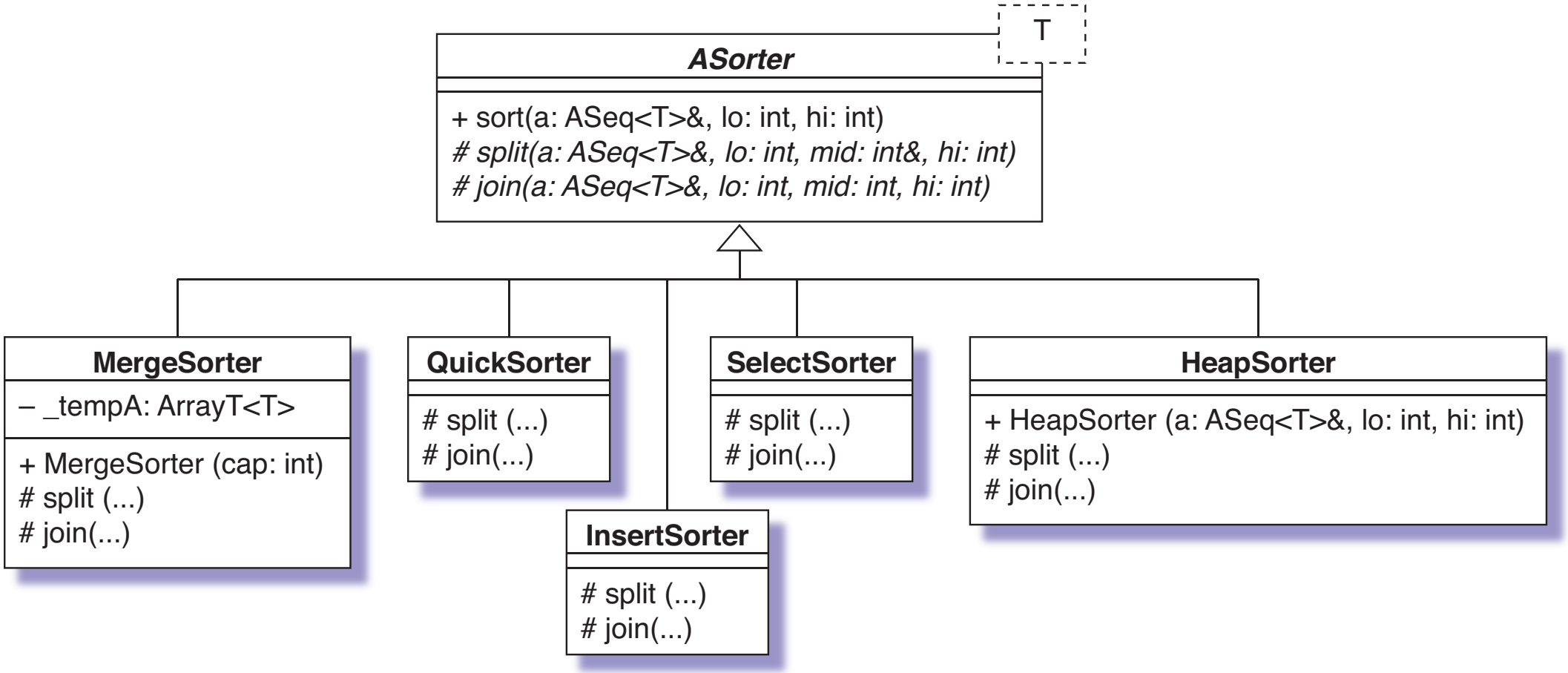
- with an invariant part, which always executes the same way,

- but which depends on variant parts that are implemented differently depending on the circumstance.

Two kinds of methods in the Template Method Pattern:

- The template method
- Primitive operations

Figure 4.6

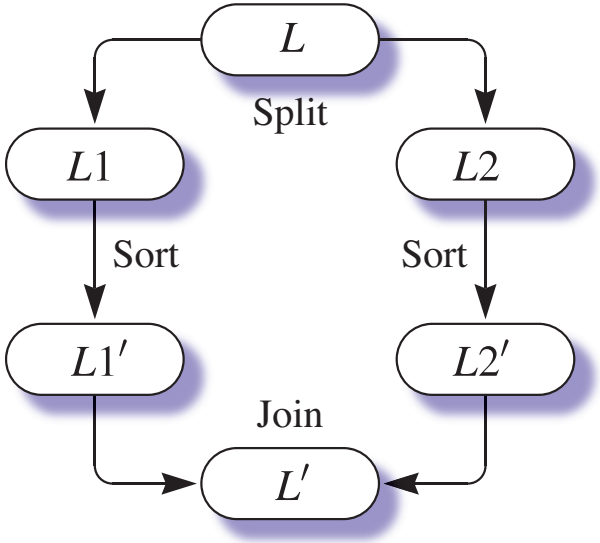

T
ASorter
+ sort(a: ASeq<T>&, lo: int, hi: int)
# split(a: ASeq<T>&, lo: int, mid: int&, hi: int)
# join(a: ASeq<T>&, lo: int, mid: int, hi: int)
MergeSorter
– _tempA: ArrayT<T>
+ MergeSorter (cap: int)
# split (...)
# join(...)
QuickSorter
split (...)
join(...)
InsertSorter
split (...)
join(...)
SelectSorter
split (...)
join(...)
HeapSorter
+ HeapSorter (a: ASeq<T>&, lo: int, hi: int)
# split (...)
# join(...)

```
template<class T>
class ASorter {

public:
    void sort(ASeq<T> &a, int lo, int hi);
    // Pre: 0 <= lo && hi < a.cap().
    // Post: sorted(a[lo..hi]).

    virtual ~ASorter() = default;
    // Virtual destructor necessary for subclassing.

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) = 0;
    // Pre: lo < hi.
    // Post: lo < mid <= hi.

    virtual void join(ASeq<T> &a, int lo, int mid, int hi) = 0;
    // Pre: lo < mid <= hi.
    // Pre: sorted(a[lo..mid-1]) && sorted(a[mid..hi]).
    // Post: sorted(a[lo..hi]).
};
```

```
template<class T>
void ASorter<T>::sort(ASeq<T> &a, int lo, int hi) {
    if (lo < 0 || a.cap() <= hi) {
        cerr << "ASorter<T>::sort precondition failed." << endl;
        cerr << "lo == " << lo << " a.cap() == " << a.cap()
             << " hi == " << hi << endl;
        throw -1;
    }
    if (lo < hi) {
        int mid;
        split(a, lo, mid, hi);
        sort(a, lo, mid - 1);
        sort(a, mid, hi);
        join(a, lo, mid, hi);
    }
}
```

Figure 4.7

```
template<class T>
void ASorter<T>::sort(ASeq<T> &a, int lo, int hi) {
    if (lo < 0 || a.cap() <= hi) {
        cerr << "ASorter<T>::sort precondition failed." << endl;
        cerr << "lo == " << lo << " a.cap() == " << a.cap()
             << " hi == " << hi << endl;
        throw -1;
    }
    if (lo < hi) {
        int mid;
        split(a, lo, mid, hi);
        sort(a, lo, mid - 1);
        sort(a, mid, hi);
        join(a, lo, mid, hi);
    }
}
```



Sort precondition:
$0 \leq l \wedge h < a.cap()$

```
sort(a, lo, hi);
```

Sort postcondition:
$\text{sorted}(a[l..h])$

`if` $l < h$

Sort precondition:
$0 \leq l \wedge h < a.cap()$

```
sort(a, lo, hi);
```

Sort postcondition:
$\text{sorted}(a[l..h])$

Sort precondition:
$0 \leq l \wedge h < a.cap()$

`sort(a, lo, hi);`

Sort postcondition:
$\text{sorted}(a[l..h])$

`if` $l < h$

Split precondition:
$l < h$

`split(a, lo, mid, hi);`

Split postcondition:
$l < m \leq h$

Sort precondition:
$0 \leq l \wedge h < a.cap()$

```
sort(a, lo, hi);
```

Sort postcondition:
$\text{sorted}(a[l..h])$

`if` $l < h$

Split precondition:
$l < h$

```
split(a, lo, mid, hi);
```

Split postcondition:
$l < m \leq h$

```
sort(a, lo, mid - 1);
sort(a, mid, hi);
```

Sort precondition:
$0 \leq l \wedge h < a.cap()$

```
sort(a, lo, hi);
```

Sort postcondition:
$\text{sorted}(a[l..h])$

if $l < h$

Split precondition:
$l < h$

```
split(a, lo, mid, hi);
```

Split postcondition:
$l < m \leq h$

```
sort(a, lo, mid - 1);
sort(a, mid, hi);
```

Join precondition:
$l < m \leq h \wedge \text{sorted}(a[l..m-1]) \wedge \text{sorted}(a[m..h])$

```
join(a, lo, mid, hi);
```

Join postcondition:
$\text{sorted}(a[l..h])$

Sort precondition:
$0 \leq l \wedge h < a.cap()$

```
sort(a, lo, hi);
```

Sort postcondition:
$\text{sorted}(a[l..h])$

if $l < h$

Split precondition:
$l < h$

```
split(a, lo, mid, hi);
```

Split postcondition:
$l < m \leq h$

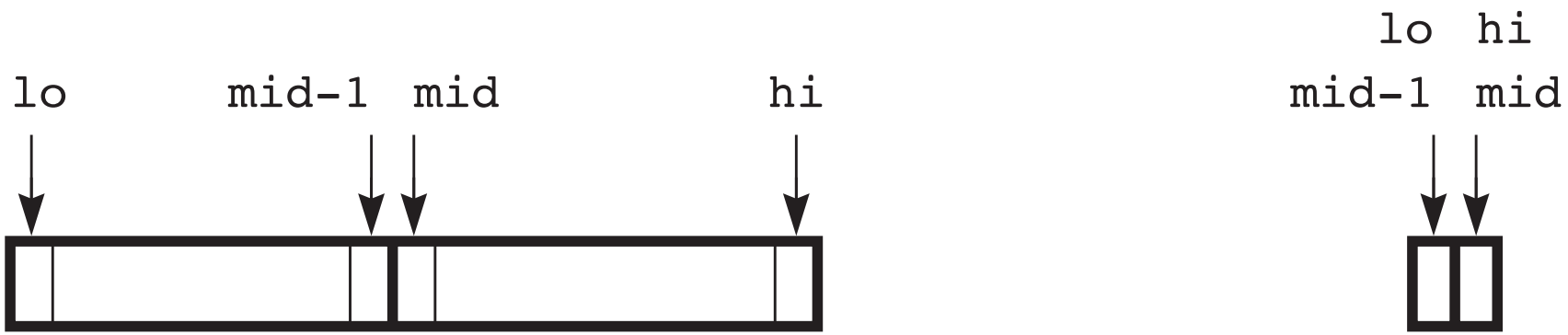
```
sort(a, lo, mid - 1);
sort(a, mid, hi);
```

Join precondition:
$l < m \leq h \wedge \text{sorted}(a[l..m-1]) \wedge \text{sorted}(a[m..h])$

```
join(a, lo, mid, hi);
```

Join postcondition:
$\text{sorted}(a[l..h])$

The `split()` primitive operation



(a) The general array `a`.

(b) Array `a` with the smallest segments.

```
virtual void split(ASeq<T> &a, int lo, int &mid, int hi) = 0;
// Pre: lo < hi.
// Post: lo < mid <= hi.
```

Using a sorter

```
shared_ptr<ASorter<int>> sorter;
```

Using a sorter

```
shared_ptr<ASorter<int>> sorter;

sorter = make_shared<QuickSorter<int>>();
```

Using a sorter

```
shared_ptr<ASorter<int>> sorter;

sorter = make_shared<QuickSorter<int>>();

ArrayT<int> array(promptIntGE("Enter array capacity", 1));
```

Using a sorter

```
shared_ptr<ASorter<int>> sorter;

sorter = make_shared<QuickSorter<int>>();

ArrayT<int> array(promptIntGE("Enter array capacity", 1));

sorter->sort(array, 0, length - 1);
```

Figure 4.8

MergeSorter

Figure 4.9

```
template<class T>
class MergeSorter : public ASorter<T> {
private:
    ArrayT<T> _tempA;

public:
    MergeSorter(int cap);

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};
```

```
template<class T>
MergeSorter<T>::MergeSorter(int cap):
    _tempA(cap) {
}

template<class T>
void MergeSorter<T>::split(ASeq<T> &, int lo, int &mid, int hi) {
// Post: mid ==(lo + hi + 1) / 2
    mid = (lo + hi + 1) / 2;
}

template<class T>
void MergeSorter<T>::join(ASeq<T> &a, int lo, int mid, int hi) {
    cerr << "MergeSorter<T>::join: Exercise for the student." << endl;
    throw -1;
}
```

MergeSorter

(a) Initial list

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
90	20	80	50	40	10	95	60	30	70

MergeSorter

(a) Initial list

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
90	20	80	50	40	10	95	60	30	70

(b) `split(a, lo, mid, hi)`

90	20	80	50	40	10	95	60	30	70

MergeSorter

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) Initial list	90	20	80	50	40	10	95	60	30	70
(b) `split(a, lo, mid, hi)`	90	20	80	50	40	10	95	60	30	70
(c) `sort(a, lo, mid - 1)`	20	40	50	80	90	10	95	60	30	70

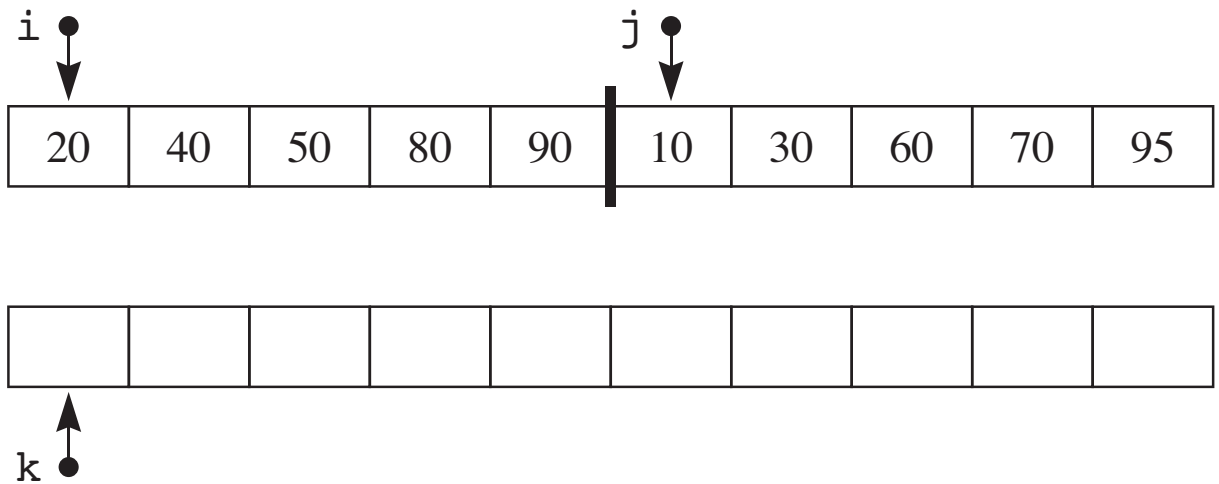
MergeSorter

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) Initial list	90	20	80	50	40	10	95	60	30	70
(b) `split(a, lo, mid, hi)`	90	20	80	50	40	10	95	60	30	70
(c) `sort(a, lo, mid - 1)`	20	40	50	80	90	10	95	60	30	70
(d) `sort(a, mid, hi)`	20	40	50	80	90	10	30	60	70	95

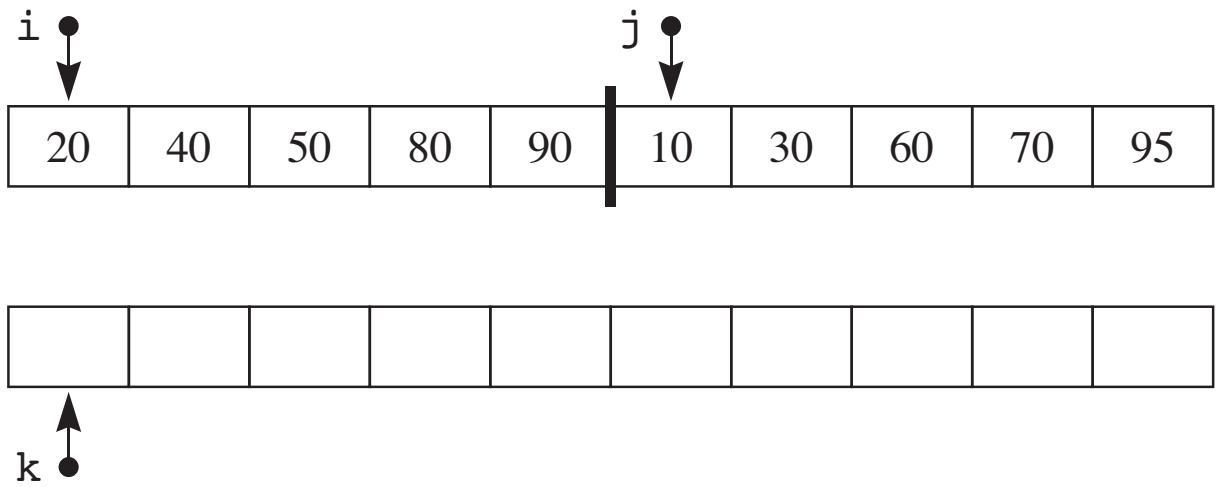
MergeSorter

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) Initial list	90	20	80	50	40	10	95	60	30	70
(b) `split(a, lo, mid, hi)`	90	20	80	50	40	10	95	60	30	70
(c) `sort(a, lo, mid - 1)`	20	40	50	80	90	10	95	60	30	70
(d) `sort(a, mid, hi)`	20	40	50	80	90	10	30	60	70	95
(e) `join(a, lo, mid, hi)`	10	20	30	40	50	60	70	80	90	95

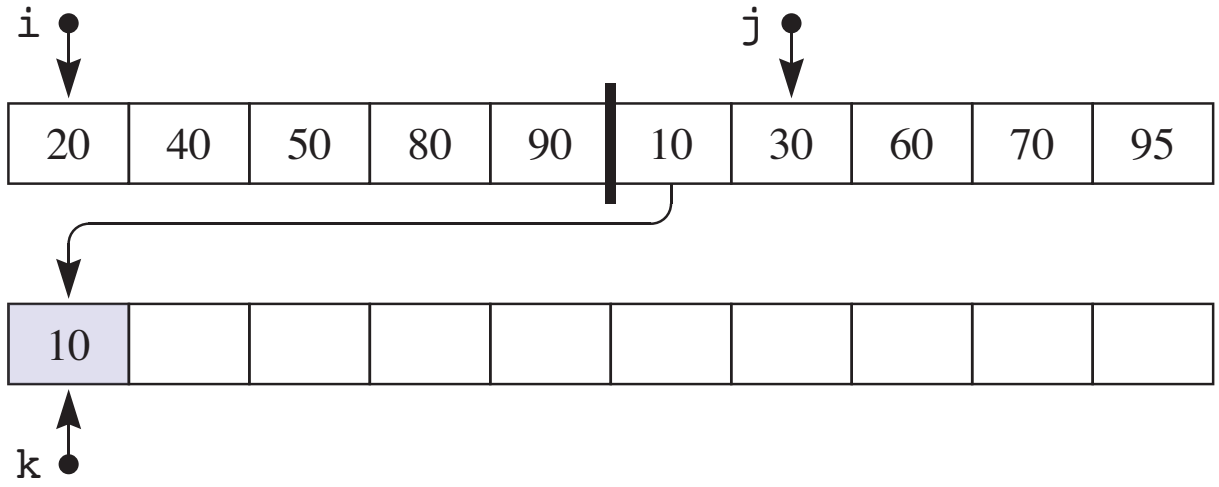
(a) Initialize `i` to the first element of the left segment, `j` to the first element of the right segment, and `k` to the first element of the segment in the temporary array.



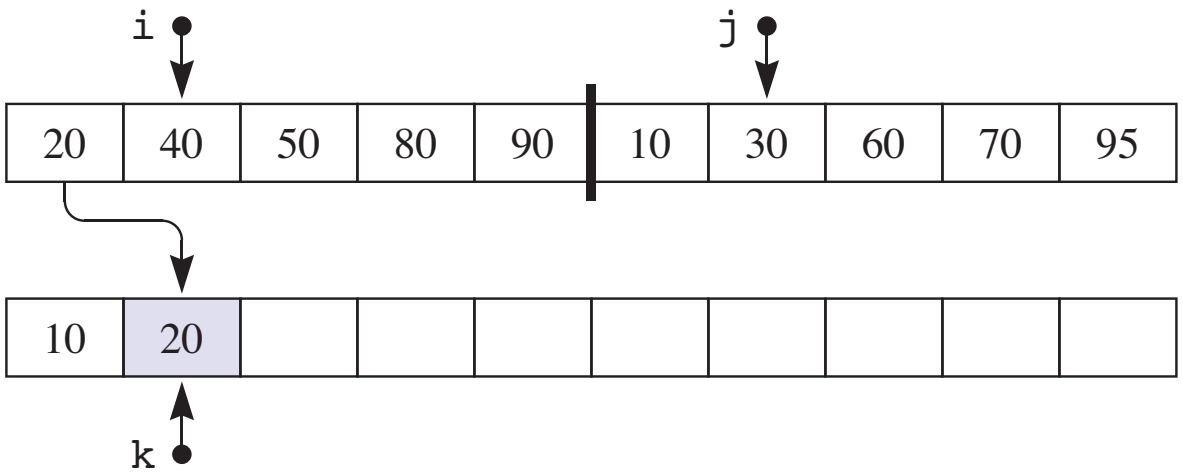
(a) Initialize `i` to the first element of the left segment, `j` to the first element of the right segment, and `k` to the first element of the segment in the temporary array.



(b) Compare 20 and 10. Because 10 is less than 20, copy `a[j]` to `_tempA[k]` and increment `j`.

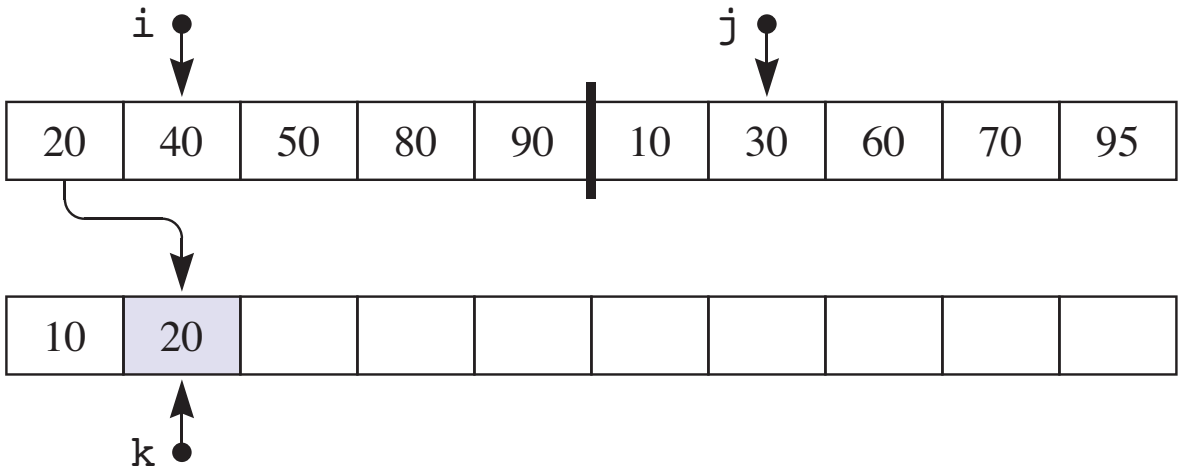


(c) Compare 20 and 30. Because 20 is less than 30, copy `a[i]` to `_tempA[k]` and increment `i`.



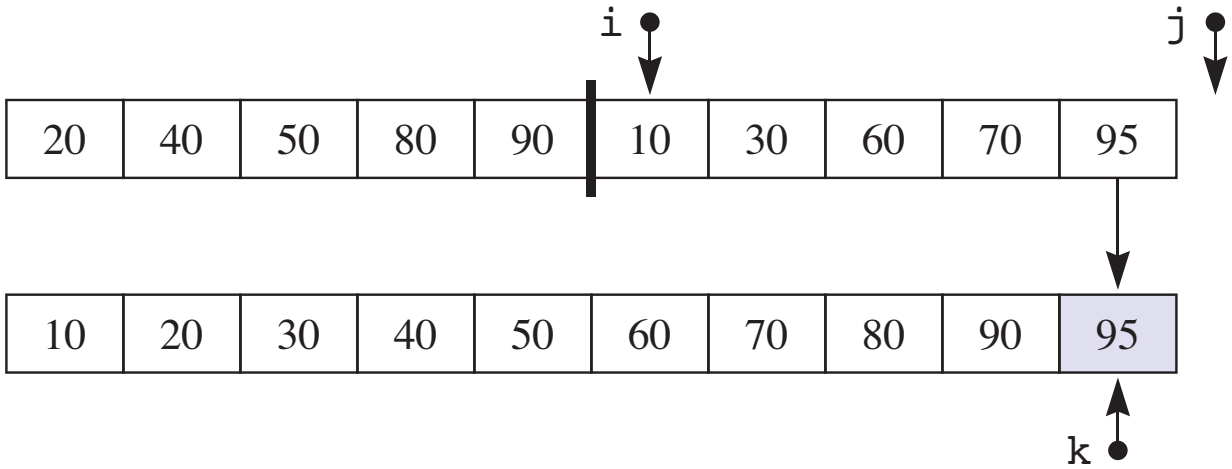
.
.

(c) Compare 20 and 30. Because 20 is less than 30, copy `a[i]` to `_tempA[k]` and increment `i`.

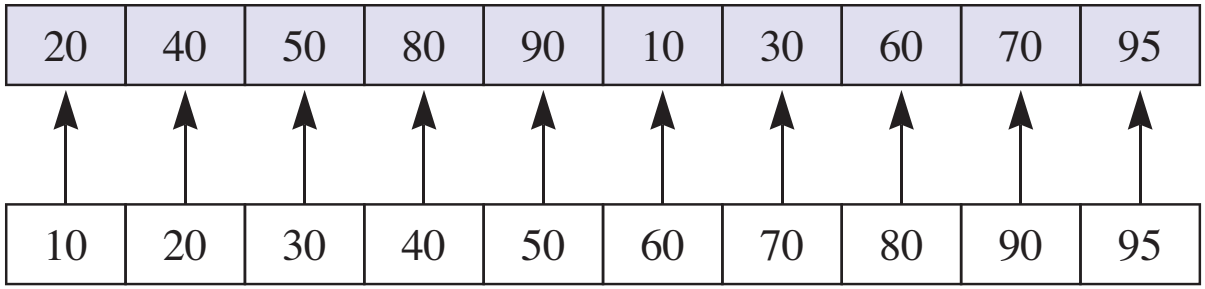


.
.
.

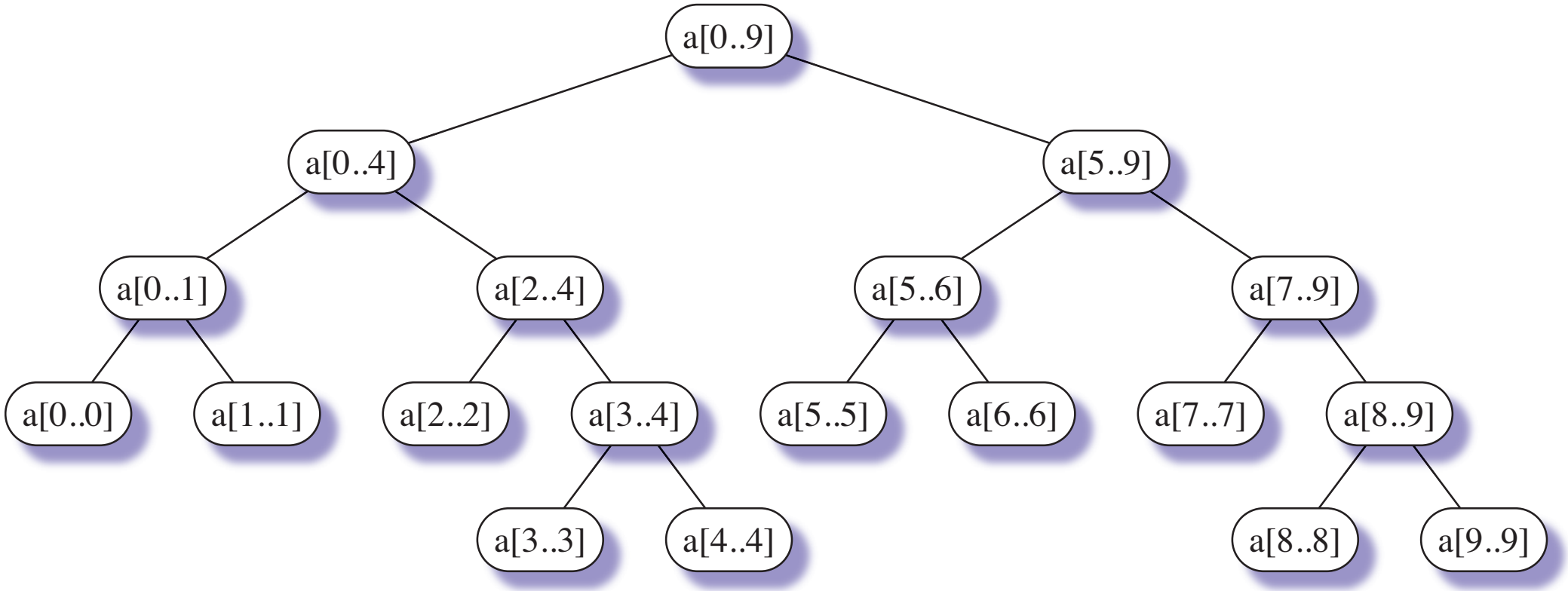
(d) Because `i` has increased past the last element of the left segment, do not compare two elements. Copy `a[j]` to `_tempA[k]` and increment `j`.



(e) Copy the segement from `_tempA` back into array `a`.



The call tree for `MergeSorter`



The performance of `MergeSorter`

The performance of `MergeSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

The performance of `MergeSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the recurrence?

The performance of `MergeSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the recurrence?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solution to the recurrence

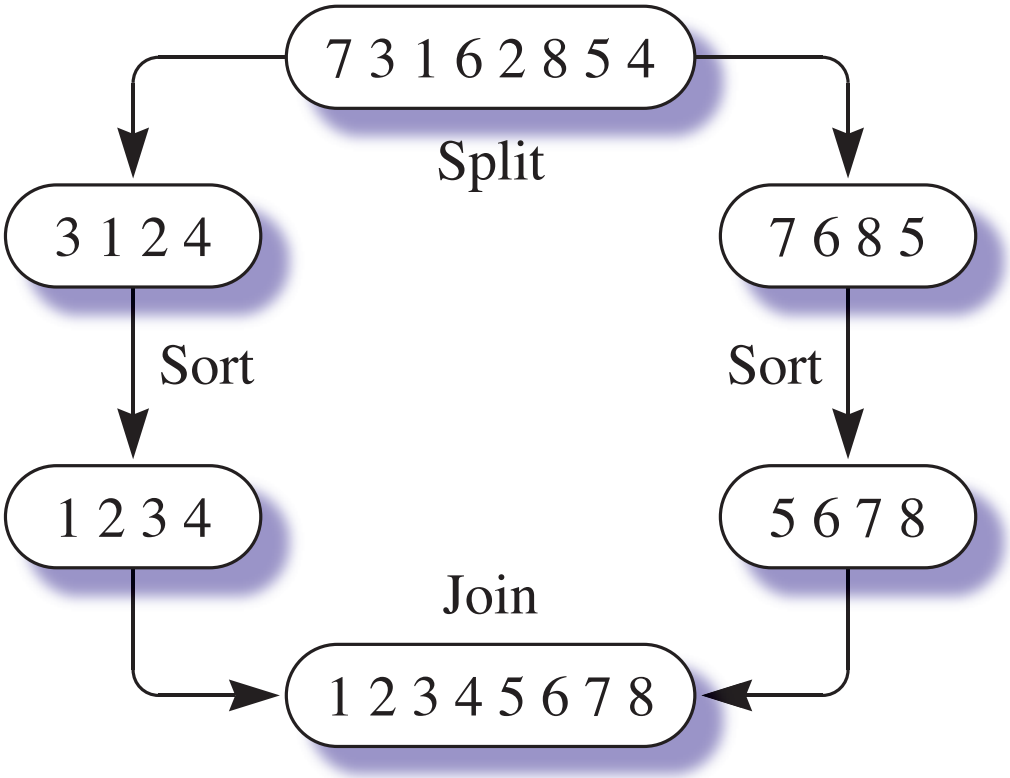
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

is

$$T(n) = \Theta(n \lg n)$$

Demo SortInt Project

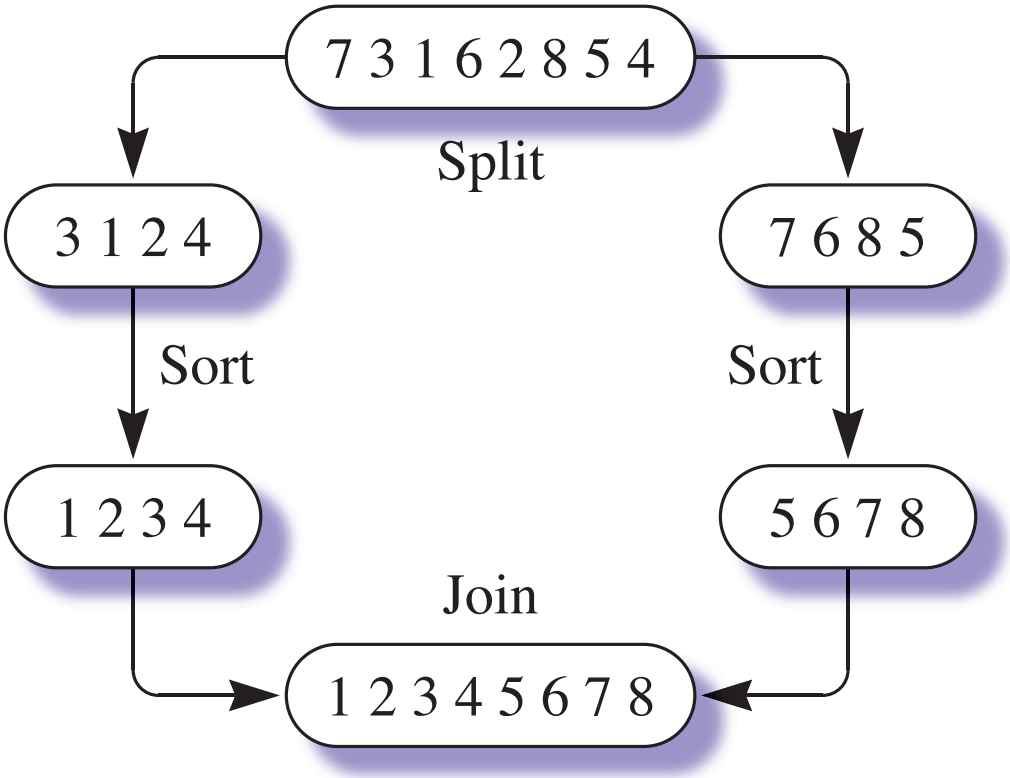
Quick Sort



(b) The quick sort algorithm.

Quick Sort

What value should you use for `key` in the split?

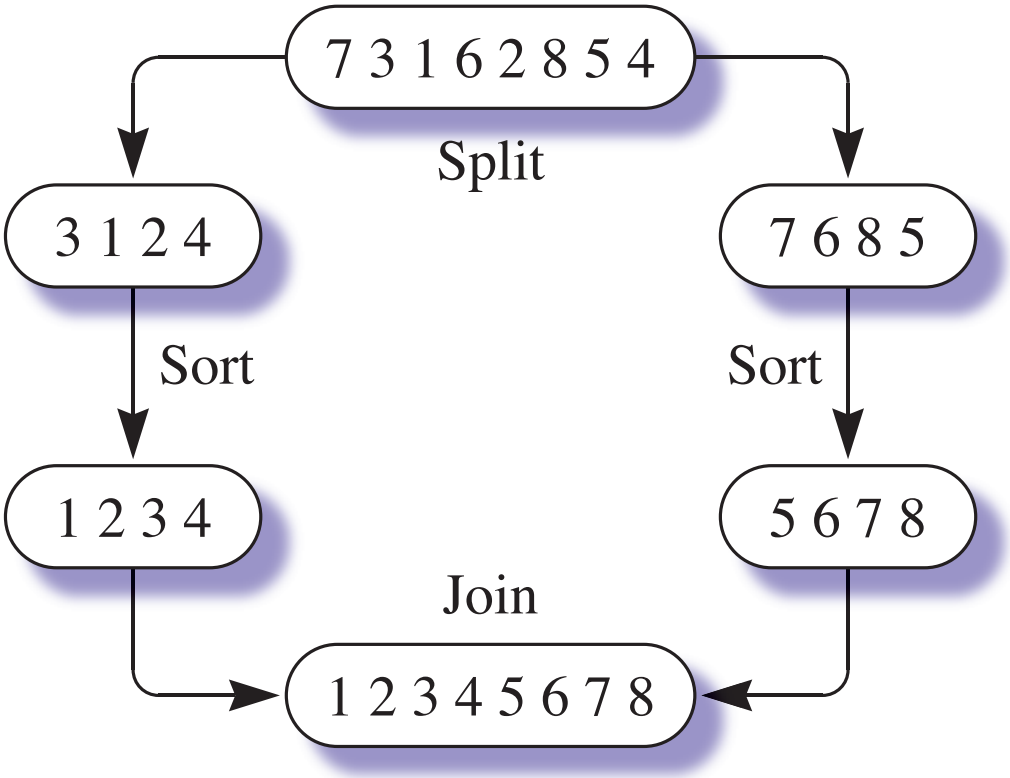


(b) The quick sort algorithm.

Quick Sort

What value should you use for `key` in the split?

The best value would be the median.



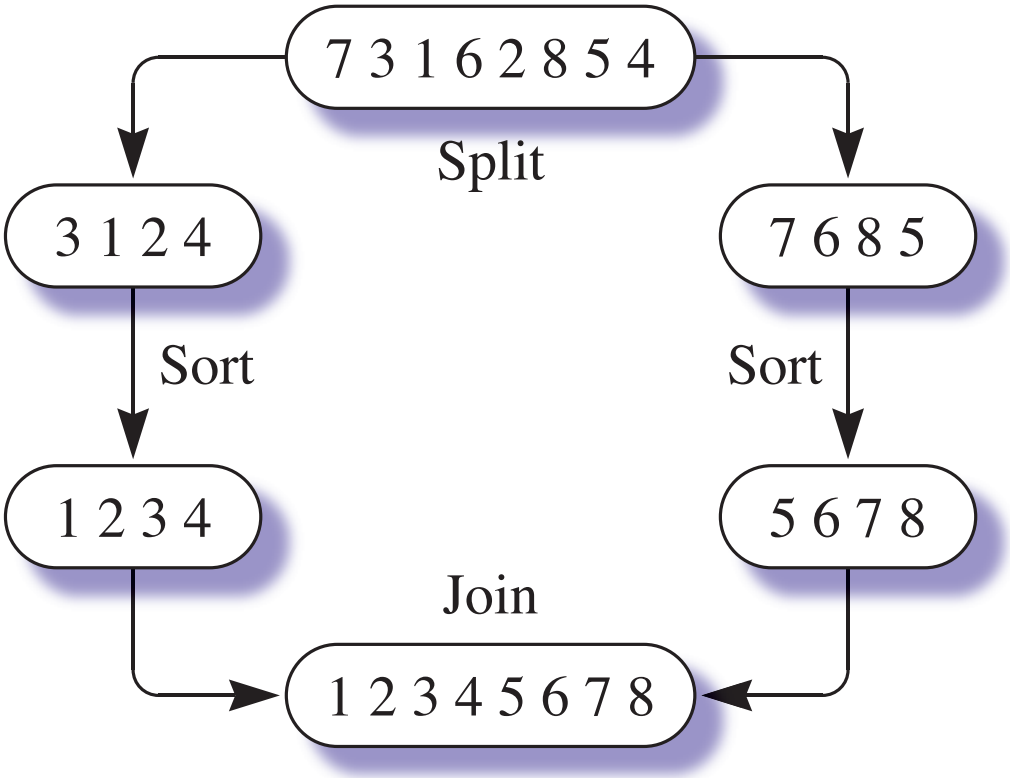
(b) The quick sort algorithm.

Quick Sort

What value should you use for `key` in the split?

The best value would be the median.

But, to find the median, you must sort the list!



(b) The quick sort algorithm.

Quick Sort

Techniques for estimating the median:

- Estimate the median of the elements in $a[l..h]$ as $a[h]$.
- Estimate the median of the elements in $a[l..h]$ as $a[r]$, where r is an integer index chosen at random from the range $l..h$.
- Estimate the median of the elements in $a[l..h]$ as the median of the three values $a[r]$, $a[s]$, and $a[t]$ where r, s, and t are integer indices chosen at random from the range $l..h$.

QuickSorter

(a) Initial list

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
90	20	80	50	40	10	95	60	30	70

QuickSorter

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) Initial list	90	20	80	50	40	10	95	60	30	70
(b) `split(a, lo, mid, hi)`	20	50	40	10	30	60	95	70	80	90

QuickSorter

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) Initial list	90	20	80	50	40	10	95	60	30	70
(b) `split(a, lo, mid, hi)`	20	50	40	10	30	60	95	70	80	90
(c) `sort(a, lo, mid - 1)`	10	20	30	40	50	60	95	70	80	90

QuickSorter

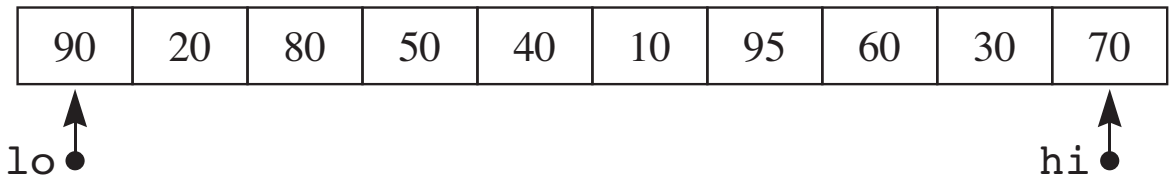
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) Initial list	90	20	80	50	40	10	95	60	30	70
(b) `split(a, lo, mid, hi)`	20	50	40	10	30	60	95	70	80	90
(c) `sort(a, lo, mid - 1)`	10	20	30	40	50	60	95	70	80	90
(d) `sort(a, mid, hi)`	10	20	30	40	50	60	70	80	90	95

QuickSorter

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
(a) Initial list	90	20	80	50	40	10	95	60	30	70
(b) `split(a, lo, mid, hi)`	20	50	40	10	30	60	95	70	80	90
(c) `sort(a, lo, mid - 1)`	10	20	30	40	50	60	95	70	80	90
(d) `sort(a, mid, hi)`	10	20	30	40	50	60	70	80	90	95
(e) `join(a, lo, mid, hi)`	10	20	30	40	50	60	70	80	90	95

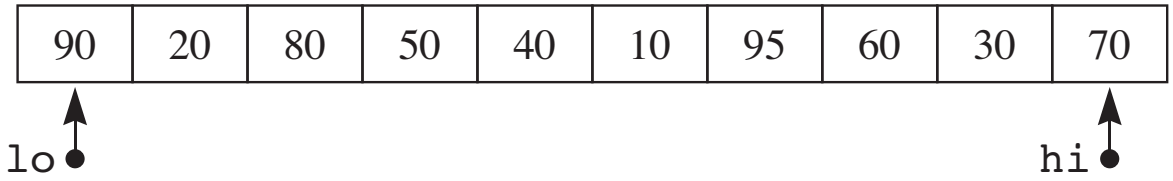
QuickSorter

(a) Initial list.

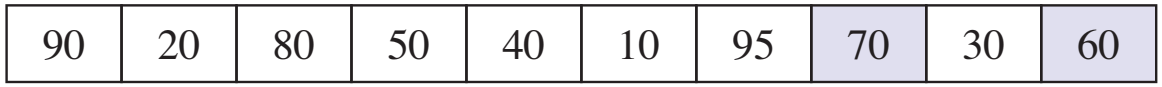


QuickSorter

(a) Initial list.

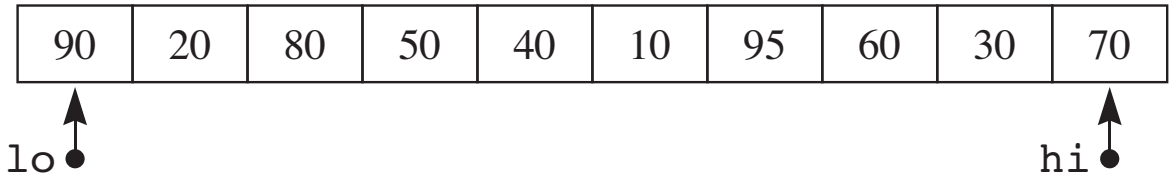


(b) Take a random sample of three elements. Swap the median of these three elements with `a[hi]`. `key=a[hi]`, which is 60.

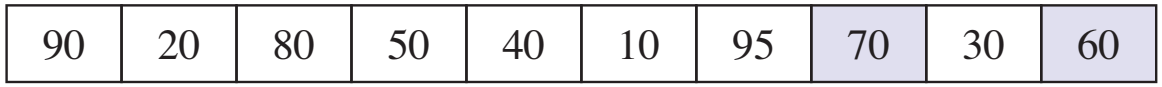


QuickSorter

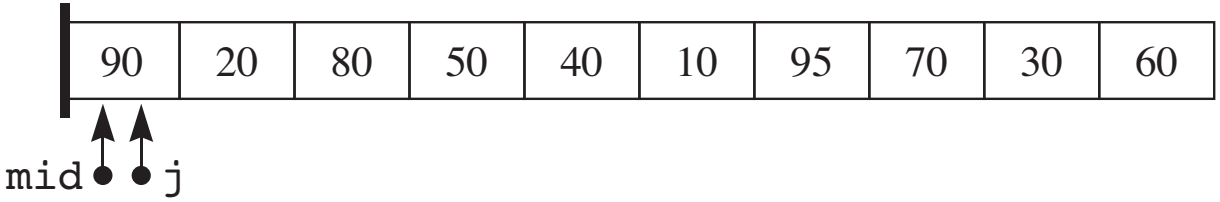
(a) Initial list.



(b) Take a random sample of three elements. Swap the median of these three elements with `a[hi]`. `key=a[hi]`, which is 60.

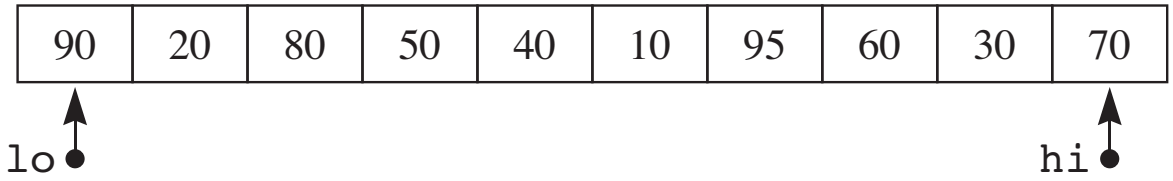


(c) Initialize `mid = lo`, `j = lo`.

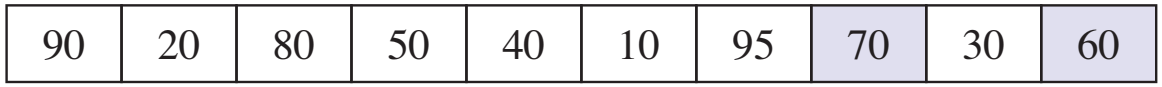


QuickSorter

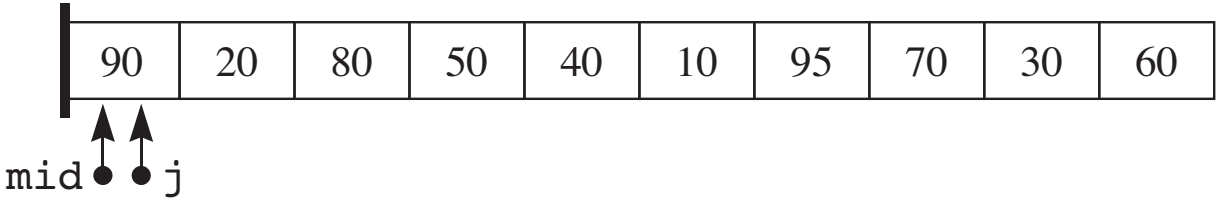
(a) Initial list.



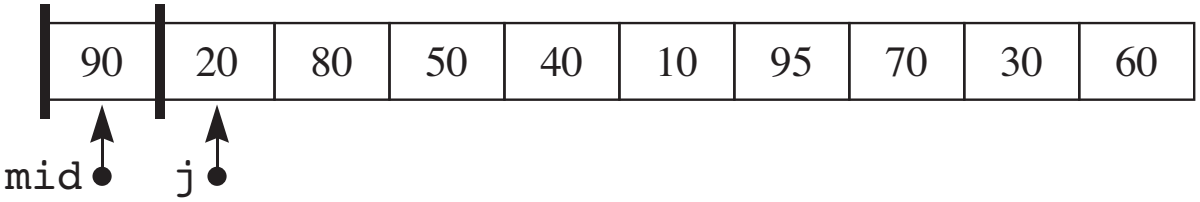
(b) Take a random sample of three elements. Swap the median of these three elements with `a[hi]`. `key=a[hi]`, which is 60.



(c) Initialize `mid = lo`, `j = lo`.

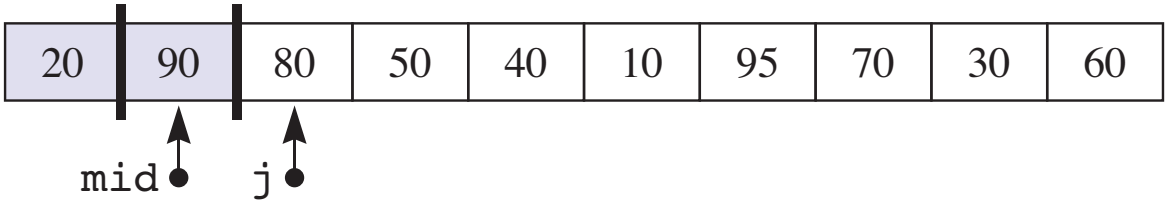


(d) Compare 90 and 60. Because 90 is greater than 60, only increment `j`.



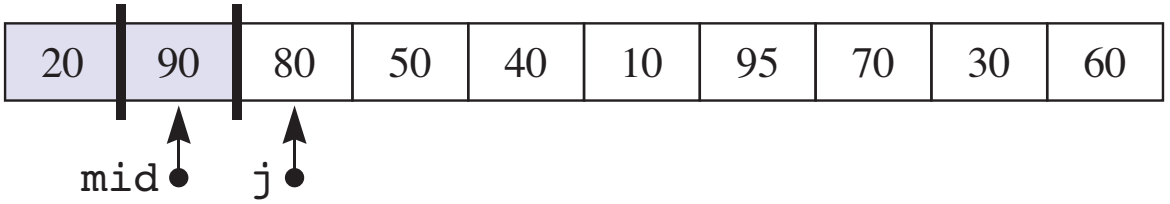
QuickSorter

(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both `j` and `mid`.

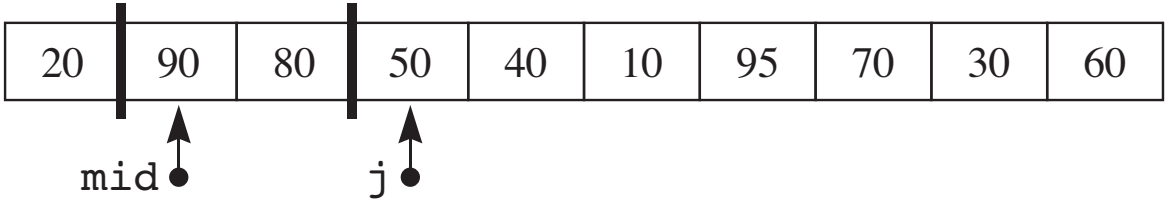


QuickSorter

(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both `j` and `mid`.

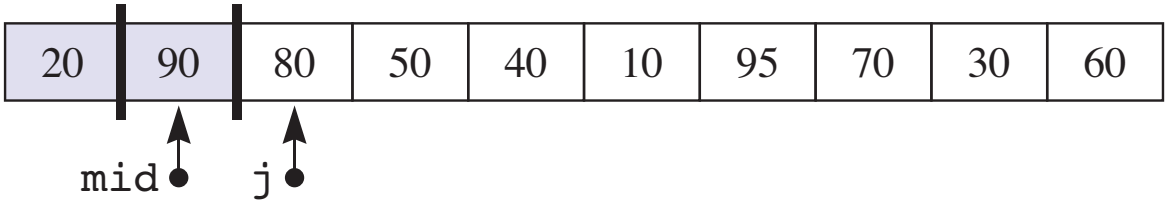


(f) Compare 80 and 60. Because 80 is greater than 60, only increment `j`.

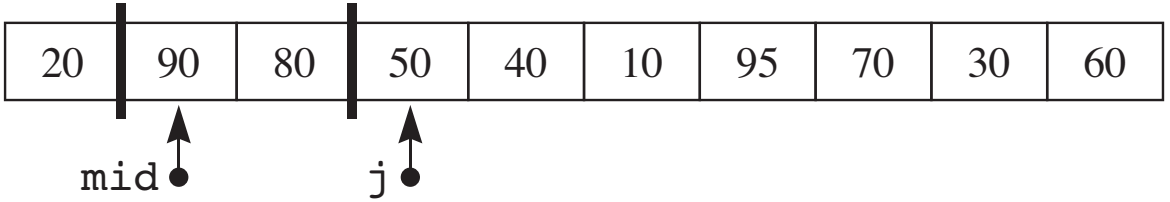


QuickSorter

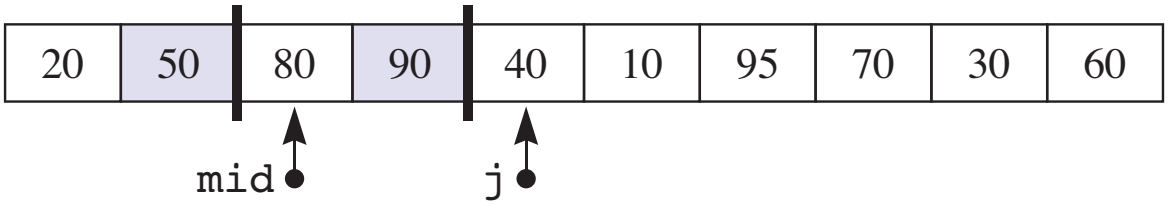
(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both `j` and `mid`.



(f) Compare 80 and 60. Because 80 is greater than 60, only increment `j`.

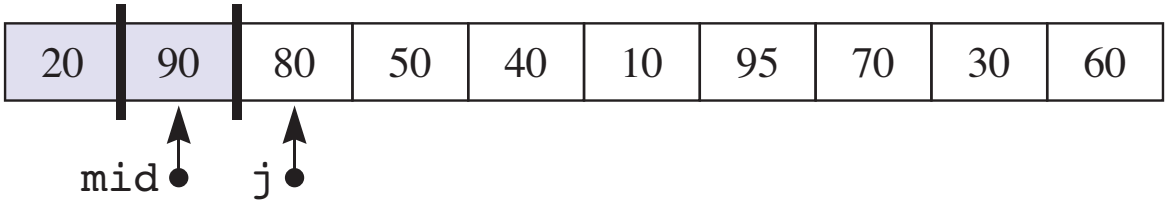


(g) Compare 50 and 60. Because 50 is less than or equal to 60, swap 50 and 90, and increment both `j` and `mid`.

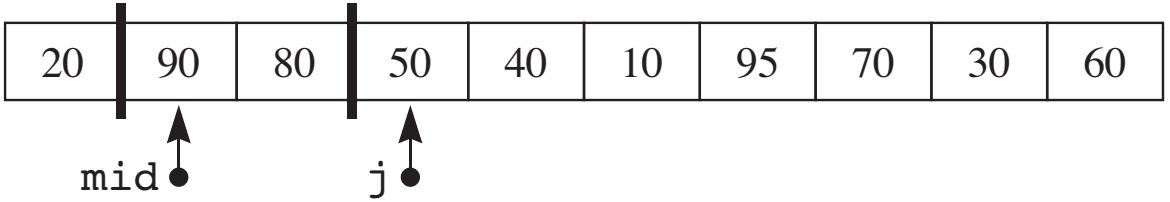


QuickSorter

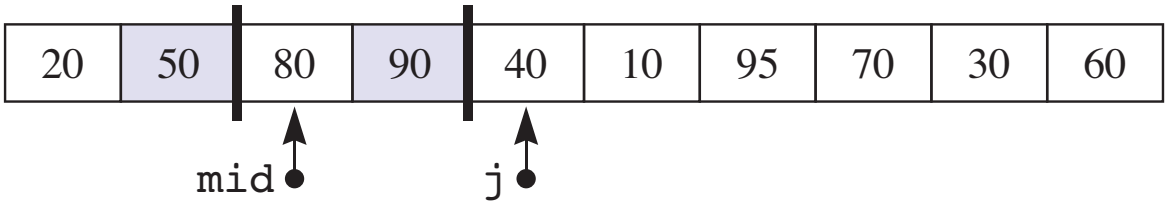
(e) Compare 20 and 60. Because 20 is less than or equal to 60, swap 20 and 90, and increment both `j` and `mid`.



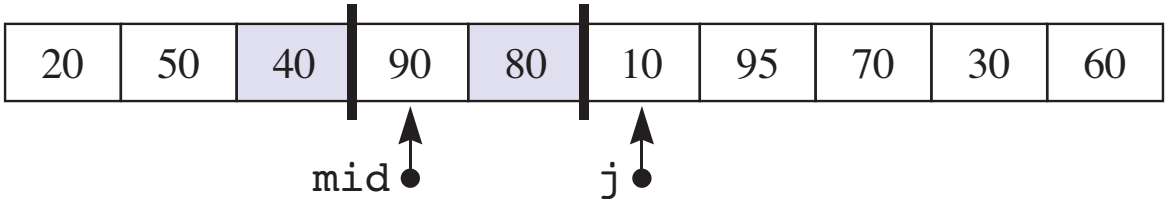
(f) Compare 80 and 60. Because 80 is greater than 60, only increment `j`.



(g) Compare 50 and 60. Because 50 is less than or equal to 60, swap 50 and 90, and increment both `j` and `mid`.

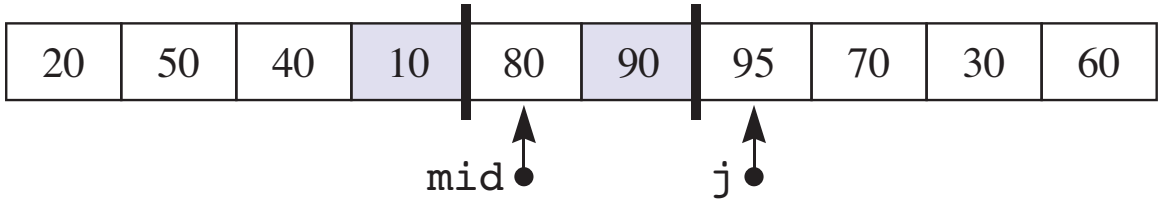


(h) Compare 40 and 60. Because 40 is less than or equal to 60, swap 40 and 80, and increment both `j` and `mid`.



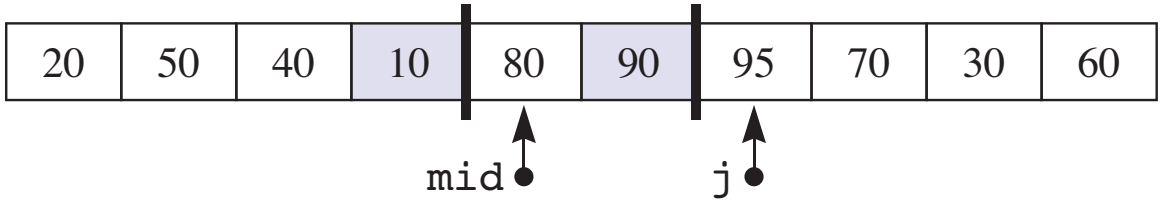
QuickSorter

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both `j` and `mid`.

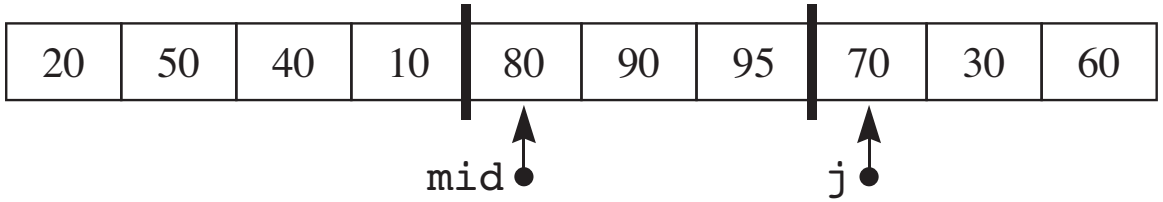


QuickSorter

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both `j` and `mid`.



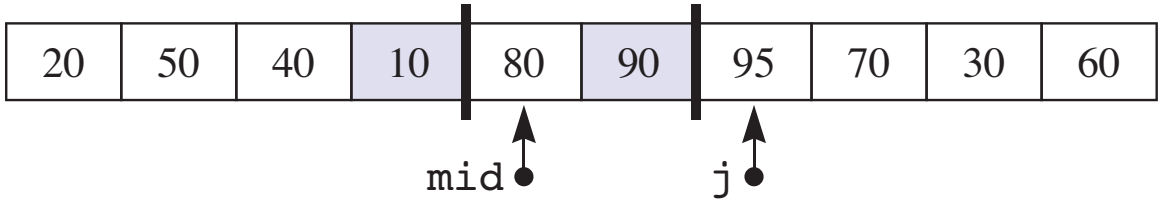
(j) Compare 95 and 60. Because 95 is greater than 60, only increment `j`.



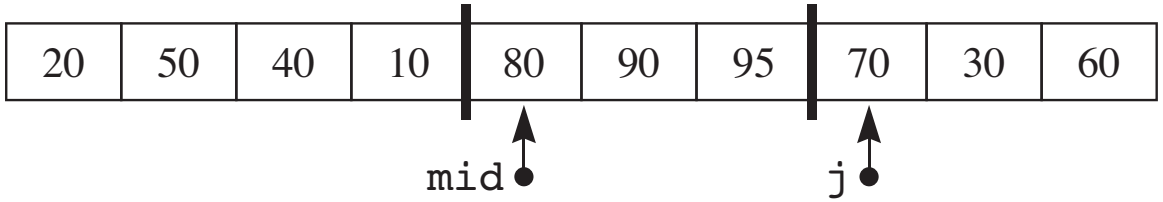
.
.

QuickSorter

(i) Compare 10 and 60. Because 10 is less than or equal to 60, swap 10 and 90, and increment both `j` and `mid`.

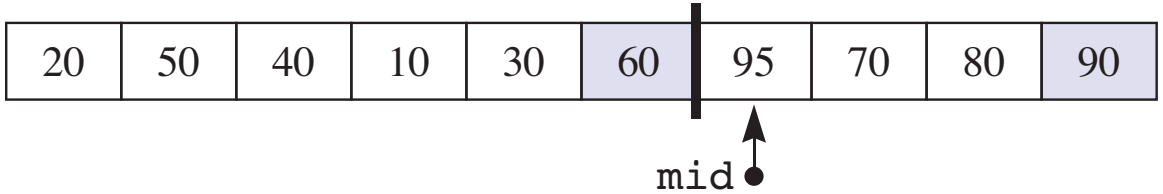


(j) Compare 95 and 60. Because 95 is greater than 60, only increment `j`.



.
.
.

(k) Compare 60 and 60. Because 60 is less than or equal to 60, swap 60 and 90, and increment `mid`.



```
#include <random>
#include "ASorter.hpp"

template<class T>
class QuickSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;

private:
    random_device rdev{};
    default_random_engine engine{rdev()};
};
```

```
template<class T>
void QuickSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
// Post: a[lo..mid-1] <= a[mid..hi]
    T temp;
    if (hi - lo > 4) {
        int mdn; // Index of the estimate of the median value.
        uniform_int_distribution<int> distr(lo, hi);
        // Find the estimate of the median.
        int mdn1 = distr(engine);
        int mdn2 = distr(engine);
        int mdn3 = distr(engine);
        if ((a[mdn2] <= a[mdn1] && a[mdn1] <= a[mdn3])
                || (a[mdn3] <= a[mdn1] && a[mdn1] <= a[mdn2])) {
            mdn = mdn1; // a[mdn1] is the median
        } else if ((a[mdn1] <= a[mdn2] && a[mdn2] <= a[mdn3])
                || (a[mdn3] <= a[mdn2] && a[mdn2] <= a[mdn1])) {
            mdn = mdn2; // a[mdn2] is the median
        } else {
            mdn = mdn3; // a[mdn3] is the median
        }
```

Additional postcondition

```
        // Swap the estimate of the median with a[hi].
        temp = a[mdn];
        a[mdn] = a[hi];
        a[hi] = temp;
    }
    // Now do the split.
    T key = a[hi];
    mid = lo;
    for (int j = lo; j <= hi; j++) {
        if (a[j] <= key) {
            temp = a[mid];
            a[mid] = a[j];
            a[j] = temp;
            mid++;
        }
    }
    mid = hi < mid ? hi : mid; // If a[hi] contains the maximum element.
}
```

Figure 4.16

```
template<class T>
void QuickSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
```

There are two postconditions

There are two postconditions

$a[l..m-1] \leq a[m..h]$

From `split()` in QuickSorter

There are two postconditions

$a[l..m-1] \leq a[m..h]$ From `split()` in QuickSorter

$l < m \leq h.$ From general `split()`

```
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++) {
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
mid = hi < mid ? hi : mid;
```

```
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++) {
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
mid = hi < mid ? hi : mid;
```

Guarantees

$$a[l..m-1] \leq a[m..h]$$

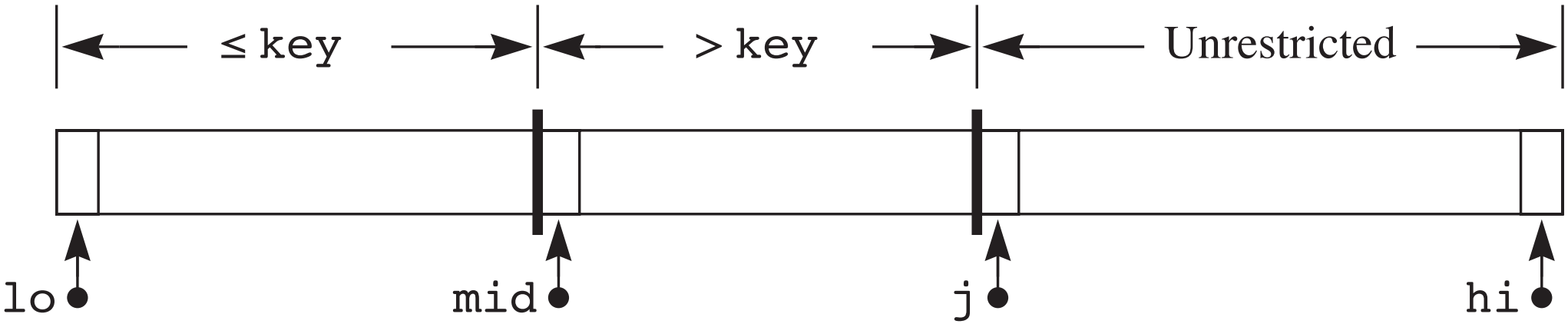
```
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++) {
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
mid = hi < mid ? hi : mid;
```

Guarantees

$$a[l..m-1] \leq a[m..h]$$

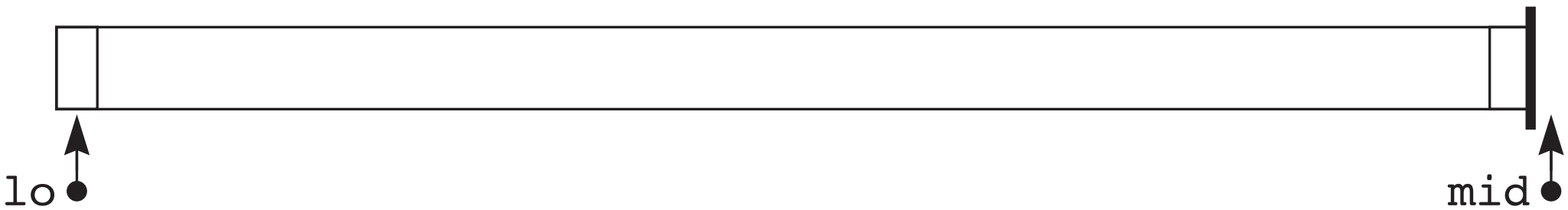
Guarantees

$$l < m \leq h.$$

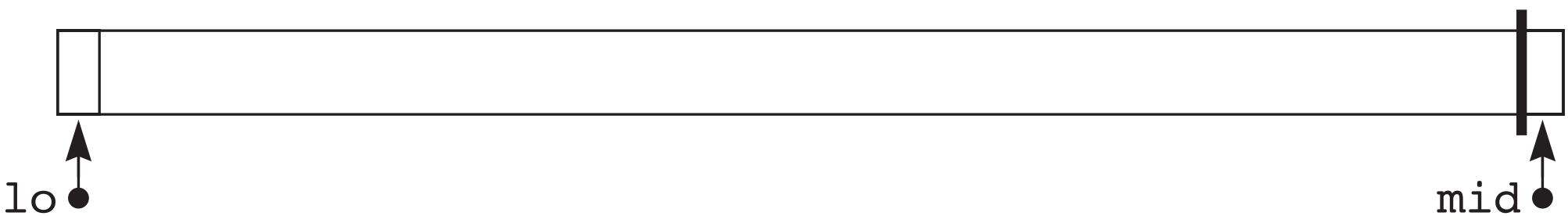


(a) The three regions during execution of the loop.

(b) The two regions after normal termination of the loop.



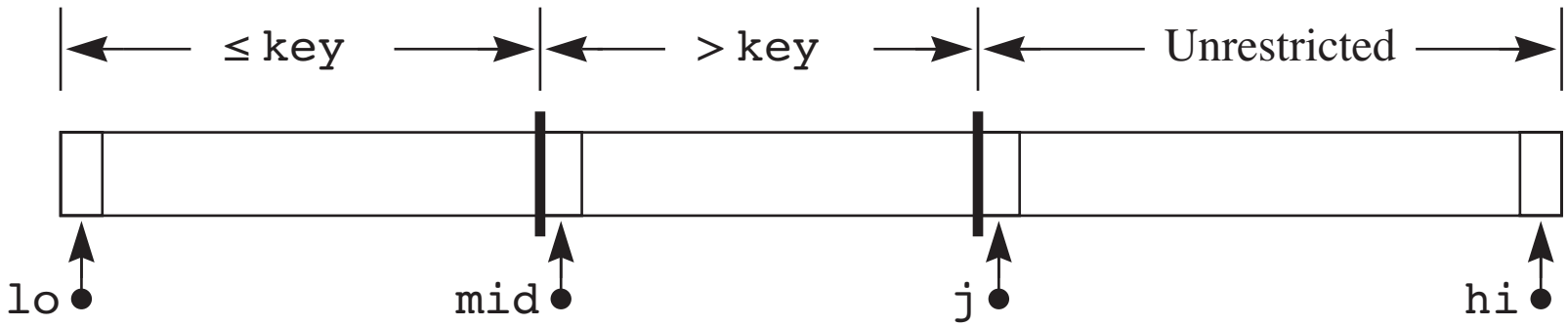
(c) After termination of the loop if `a[hi]` contains the maximum element.



(d) After adjustment of `mid` if `a[hi]` contains the maximum element.

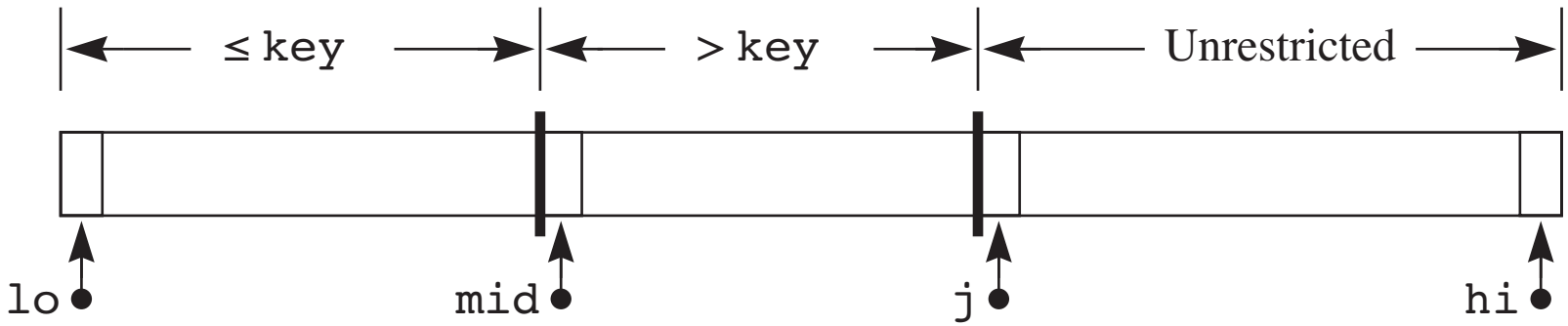
```
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++)
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
```

```
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++)
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
```



(a) The three regions during execution of the loop.

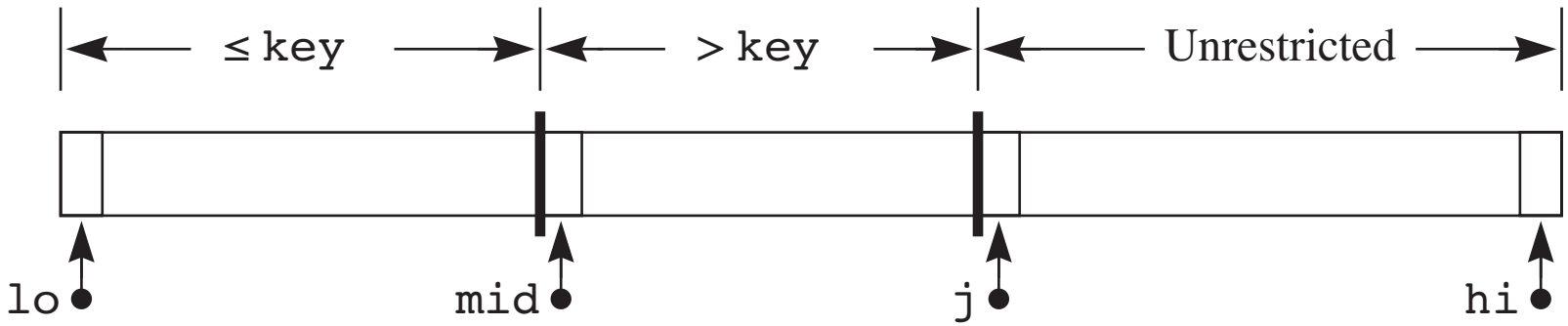
```
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++)
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
```



(a) The three regions during execution of the loop.

What is the loop invariant?

```
T key = a[hi];
mid = lo;
for (int j = lo; j <= hi; j++)
    if (a[j] <= key) {
        temp = a[mid];
        a[mid] = a[j];
        a[j] = temp;
        mid++;
    }
}
```



(a) The three regions during execution of the loop.

What is the loop invariant?

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$\begin{aligned} & (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\ = \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \end{aligned}$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
& (\forall i \mid l \leq i < l : a[i] \leq k) \wedge (\forall i \mid l \leq i < l : a[i] > k)
\end{aligned}
$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
& (\forall i \mid l \leq i < l : a[i] \leq k) \wedge (\forall i \mid l \leq i < l : a[i] > k) \\
= \quad & \langle \text{Math} \rangle
\end{aligned}
$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
& (\forall i \mid l \leq i < l : a[i] \leq k) \wedge (\forall i \mid l \leq i < l : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid false : a[i] \leq k) \wedge (\forall i \mid false : a[i] > k)
\end{aligned}
$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
& (\forall i \mid l \leq i < l : a[i] \leq k) \wedge (\forall i \mid l \leq i < l : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid false : a[i] \leq k) \wedge (\forall i \mid false : a[i] > k) \\
= \quad & \langle \text{Empty range rule} \rangle
\end{aligned}
$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
& (\forall i \mid l \leq i < l : a[i] \leq k) \wedge (\forall i \mid l \leq i < l : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid false : a[i] \leq k) \wedge (\forall i \mid false : a[i] > k) \\
= \quad & \langle \text{Empty range rule} \rangle \\
& true \wedge true
\end{aligned}
$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
& (\forall i \mid l \leq i < l : a[i] \leq k) \wedge (\forall i \mid l \leq i < l : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid false : a[i] \leq k) \wedge (\forall i \mid false : a[i] > k) \\
= \quad & \langle \text{Empty range rule} \rangle \\
& true \wedge true \\
= \quad & \langle \text{Idempotency of } \wedge \rangle
\end{aligned}
$$

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
& (\forall i \mid l \leq i < l : a[i] \leq k) \wedge (\forall i \mid l \leq i < l : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid false : a[i] \leq k) \wedge (\forall i \mid false : a[i] > k) \\
= \quad & \langle \text{Empty range rule} \rangle \\
& true \wedge true \\
= \quad & \langle \text{Idempotency of } \wedge \rangle \\
& true \quad \blacksquare
\end{aligned}
$$

The invariant is maintained with each execution of the loop.

The invariant is maintained with each execution of the loop.

See text

The invariant is maintained with each execution of the loop.

See text

The loop terminates.

The invariant is maintained with each execution of the loop.

See text

The loop terminates.

See text

The postcondition holds at the end of the loop.

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{The final value of } j \text{ is } h+1 \rangle
\end{aligned}
$$

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{The final value of } j \text{ is } h+1 \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < h+1 : a[i] > k)
\end{aligned}
$$

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{The final value of } j \text{ is } h+1 \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < h+1 : a[i] > k) \\
= \quad & \langle \text{Math} \rangle
\end{aligned}
$$

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{The final value of } j \text{ is } h+1 \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < h+1 : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i \leq h : a[i] > k)
\end{aligned}
$$

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{The final value of } j \text{ is } h+1 \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < h+1 : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i \leq h : a[i] > k) \\
\Rightarrow \quad & \langle \text{Transitivity of the } < \text{ operator and logic} \rangle
\end{aligned}
$$

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k) \\
= \quad & \langle \text{The final value of } j \text{ is } h+1 \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < h+1 : a[i] > k) \\
= \quad & \langle \text{Math} \rangle \\
& (\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i \leq h : a[i] > k) \\
\Rightarrow \quad & \langle \text{Transitivity of the } < \text{ operator and logic} \rangle \\
& a[l..m-1] \leq a[m..h]
\end{aligned}
$$

which is the second postcondition of `sort()`.

The performance of `QuickSorter`

The performance of `QuickSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

The performance of `QuickSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the best case recurrence?

The performance of `QuickSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the best case recurrence?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solution to the best case recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

is

$$T(n) = \Theta(n \lg n)$$

The performance of `QuickSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the worst case recurrence?

The performance of `QuickSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the worst case recurrence?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solution to the worst case recurrence

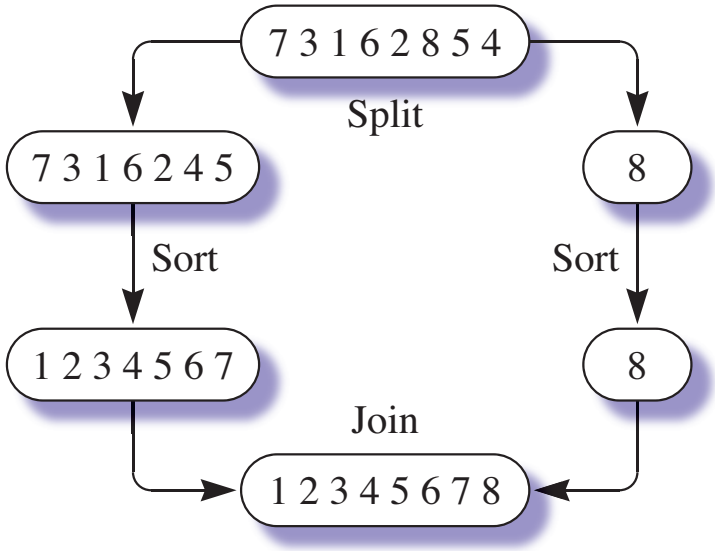
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

is

$$T(n) = \Theta(n^2)$$

SelectSorter

```
template<class T>
class SelectSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};
```

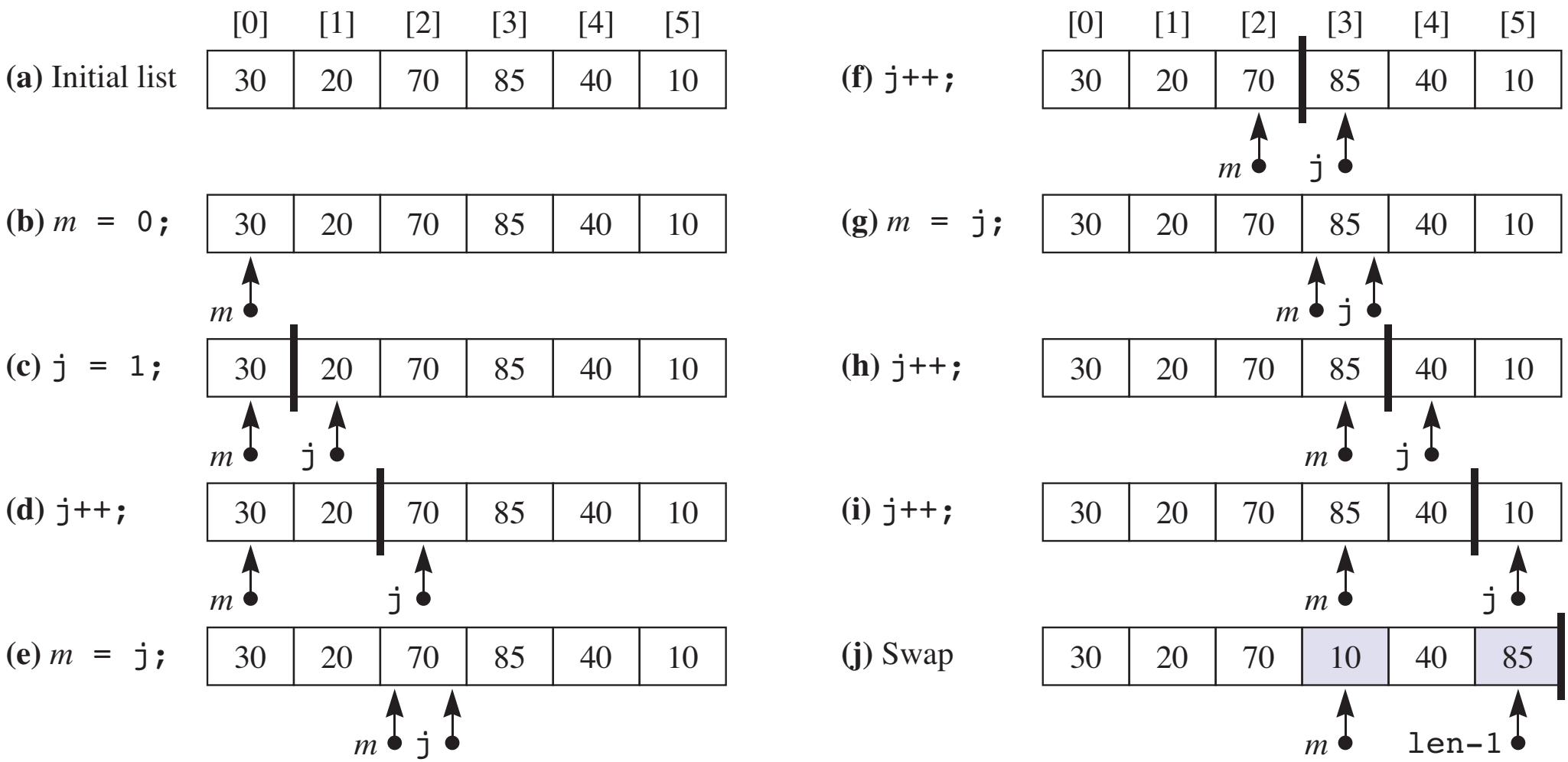


(b) The selection sort algorithm.

```
template<class T>
void SelectSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
// Post: a[hi] == max(a[lo..hi]).
// Post: mid == hi.
    int indexOfMax = lo;
    for (int i = lo + 1; i <= hi; i++) {
        if (a[indexOfMax] < a[i]) {
            indexOfMax = i;
        }
    }
    T temp = a[hi];
    a[hi] = a[indexOfMax];
    a[indexOfMax] = temp;
    mid = hi;
}

template<class T>
void SelectSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
```

```
template<class T>
void largestLast(ASeq<T> &a, int len) {
    int indexOfMax = 0;
    for (int j = 1; j < len; j++) {
        if (a[indexOfMax] < a[j]) {
            indexOfMax = j;
        }
    }
    T temp = a[len - 1];
    a[len - 1] = a[indexOfMax];
    a[indexOfMax] = temp;
}
```

Figure 3.1


[0] [1] [2] [3] [4] [5]
(a) Initial list
30 20 70 85 40 10
(b) m = 0;
30 20 70 85 40 10
m
(c) j = 1;
30 20 70 85 40 10
m j
(d) j++;
30 20 70 85 40 10
m j
(e) m = j;
30 20 70 85 40 10
m j
(f) j++;
30 20 70 85 40 10
m j
(g) m = j;
30 20 70 85 40 10
m j
(h) j++;
30 20 70 85 40 10
m j
(i) j++;
30 20 70 85 40 10
m j
(j) Swap
30 20 70 10 40 85
m len-1

The performance of `SelectSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the recurrence?

The performance of `SelectSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the recurrence?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solution to the recurrence

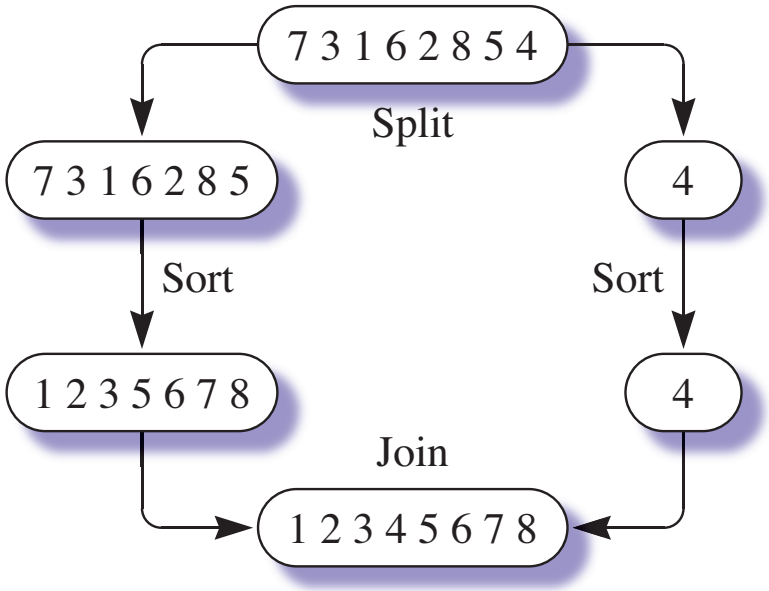
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

is

$$T(n) = \Theta(n^2)$$

InsertSorter

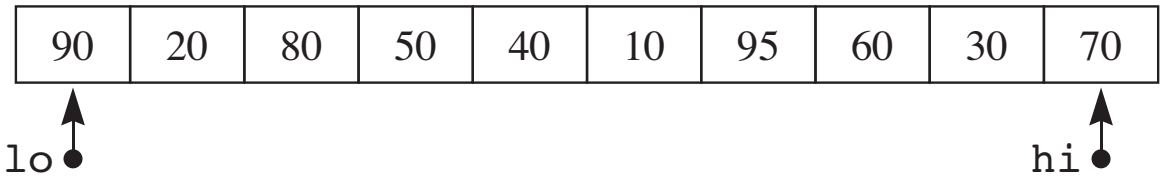
```
template<class T>
class InsertSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T>&, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T>&, int lo, int mid, int hi) override;
};
```



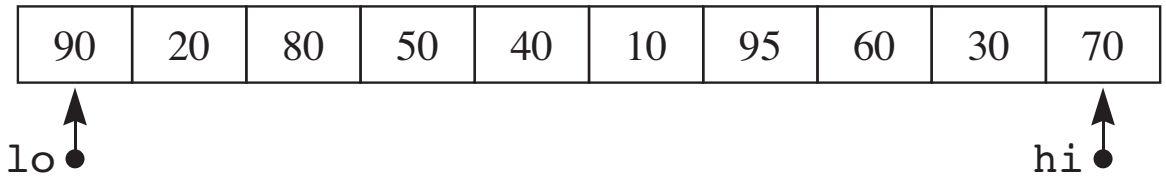
(a) The insertion sort algorithm.

```
template<class T>
void InsertSorter<T>::split(ASeq<T> &, int , int &mid, int hi) {
// Post: mid == hi.
    mid = hi;
}

template<class T>
void InsertSorter<T>::join(ASeq<T> &a, int lo, int mid, int hi) {
// Pre: mid == hi && sorted(a[lo..hi - 1]).
// Post: sorted(a[lo..hi]).
    cerr << "InsertSorter<T>::join: Exercise for the student." << endl;
    throw -1;
}
```

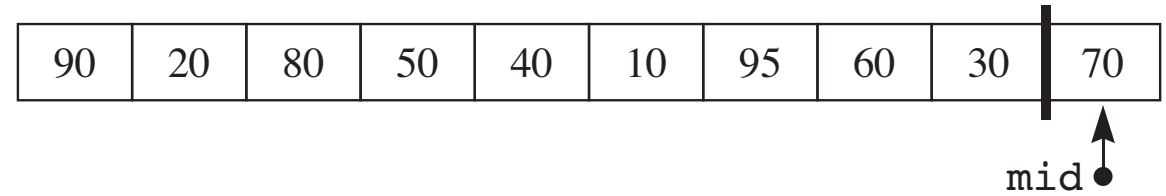
(a) Initial list.



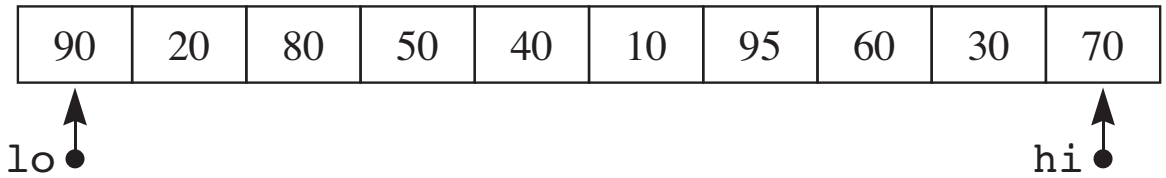
(a) Initial list.



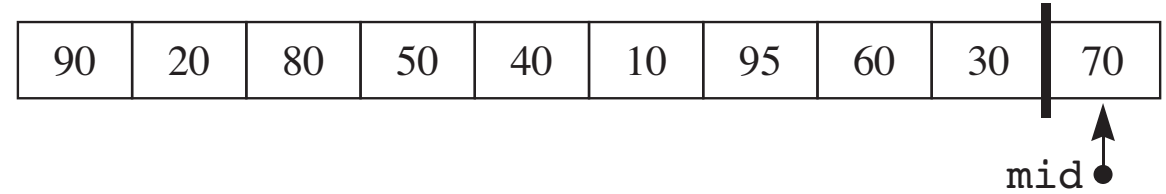
(b) `split(a, lo, mid, hi)`
Sets `mid` with `mid = hi`.



(a) Initial list.



(b) `split(a, lo, mid, hi)`
Sets `mid` with `mid = hi`.



(c) `sort(a, lo, mid - 1)`

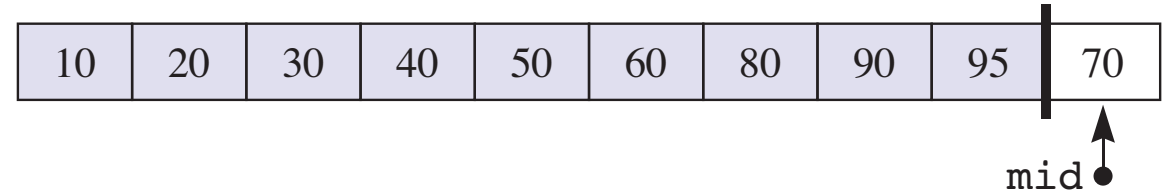
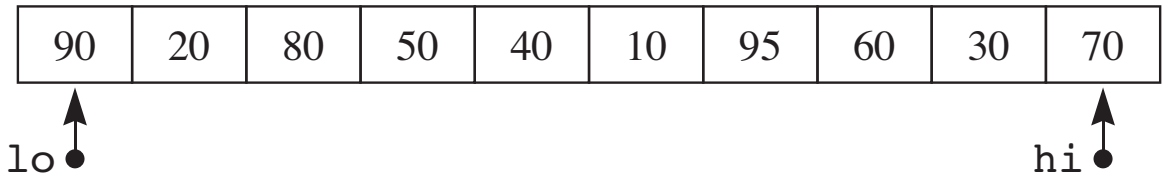
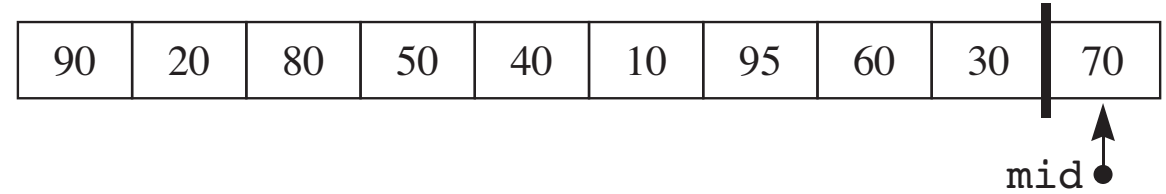


Figure 4.20

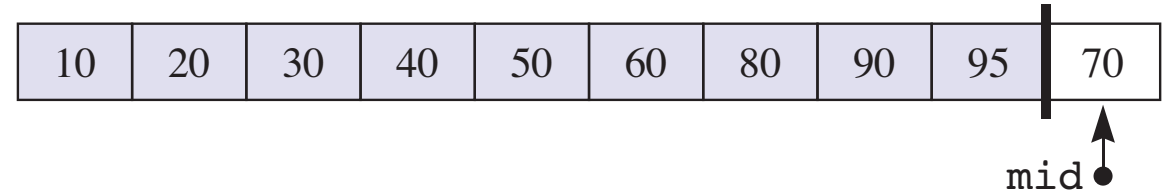
(a) Initial list.



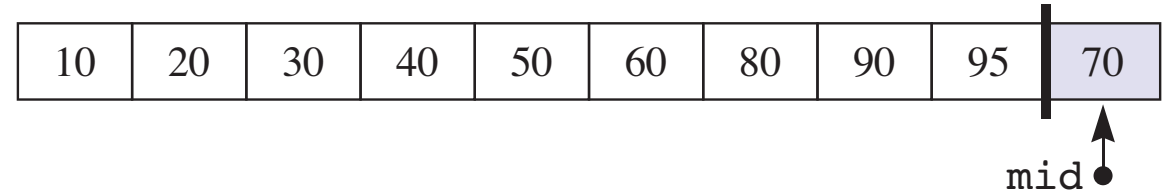
(b) `split(a, lo, mid, hi)`
Sets `mid` with `mid = hi`.



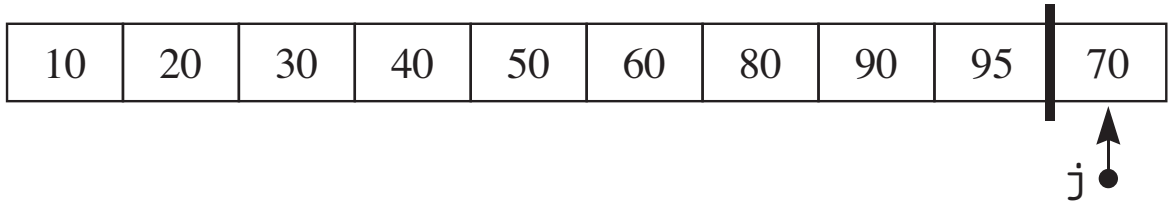
(c) `sort(a, lo, mid - 1)`



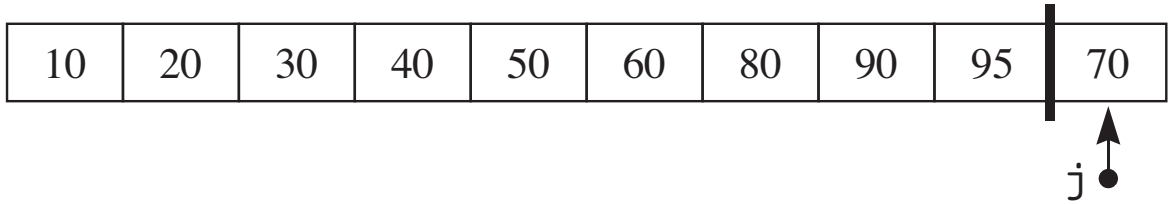
(d) `sort(a, mid, hi)`
No work is done.



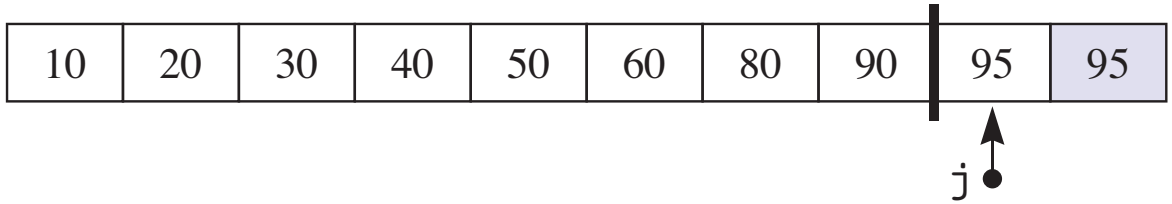
(e) `join(a, lo, mid, hi)`
Initialize `j` to `mid` and `key` to `a[mid]`, which is 70.



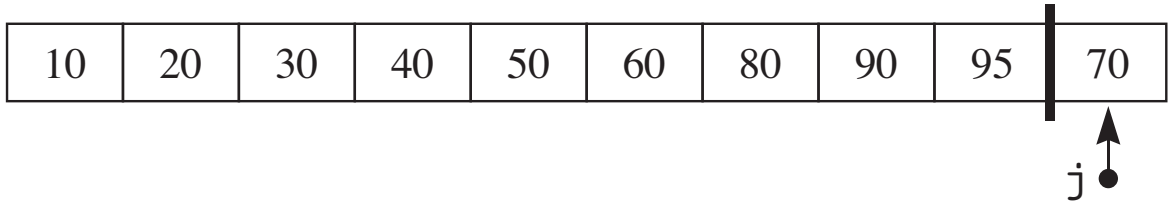
(e) `join(a, lo, mid, hi)`
Initialize `j` to `mid` and `key` to `a[mid]`, which is 70.



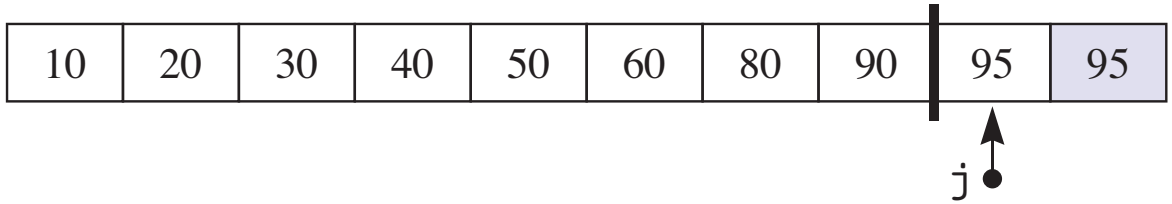
(f) Compare 70 and 95. Because 95 is greater than 70, set `a[j]` to 95 and decrement `j`.



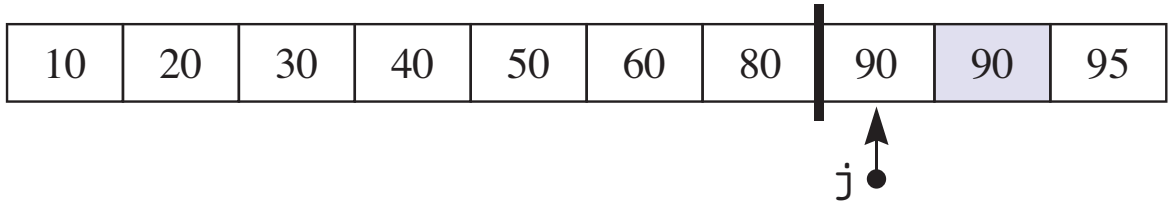
(e) `join(a, lo, mid, hi)`
Initialize `j` to `mid` and `key` to `a[mid]`, which is 70.



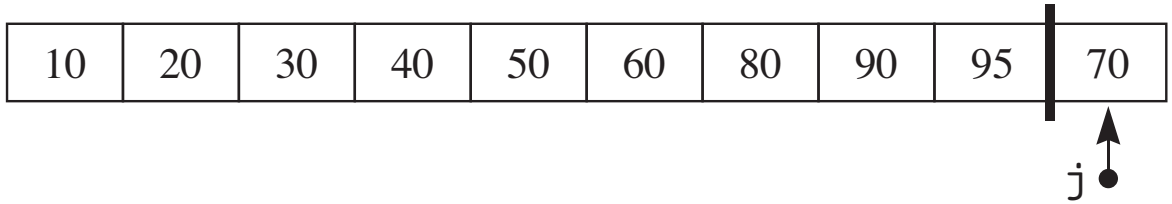
(f) Compare 70 and 95. Because 95 is greater than 70, set `a[j]` to 95 and decrement `j`.



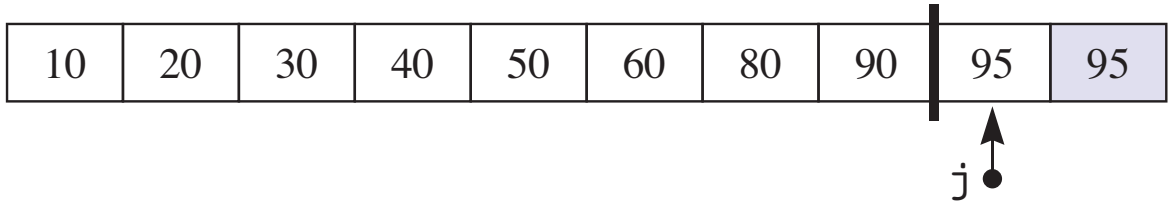
(g) Compare 70 and 90. Because 90 is greater than 70, set `a[j]` to 90 and decrement `j`.



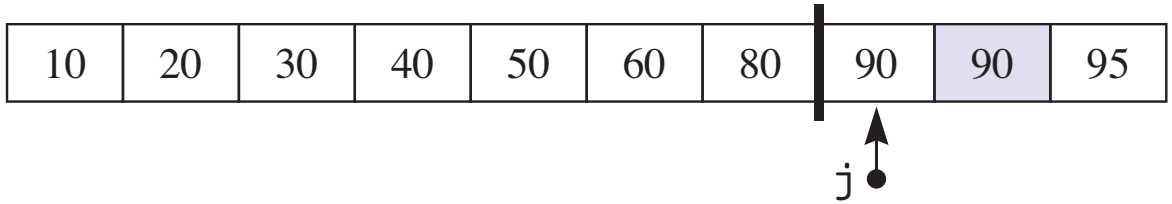
(e) `join(a, lo, mid, hi)`
Initialize `j` to `mid` and `key` to `a[mid]`, which is 70.



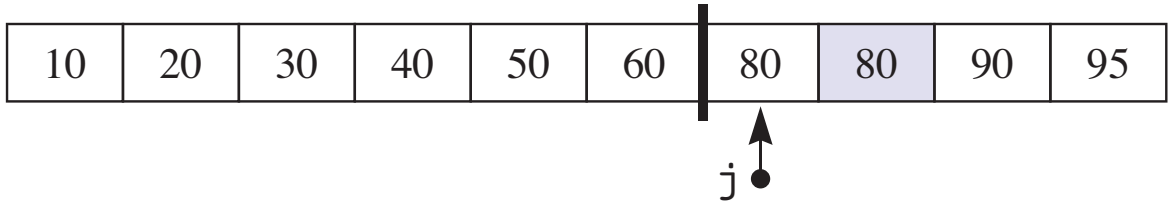
(f) Compare 70 and 95. Because 95 is greater than 70, set `a[j]` to 95 and decrement `j`.



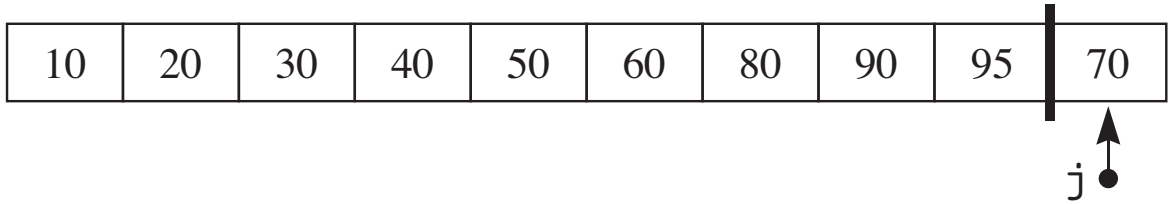
(g) Compare 70 and 90. Because 90 is greater than 70, set `a[j]` to 90 and decrement `j`.



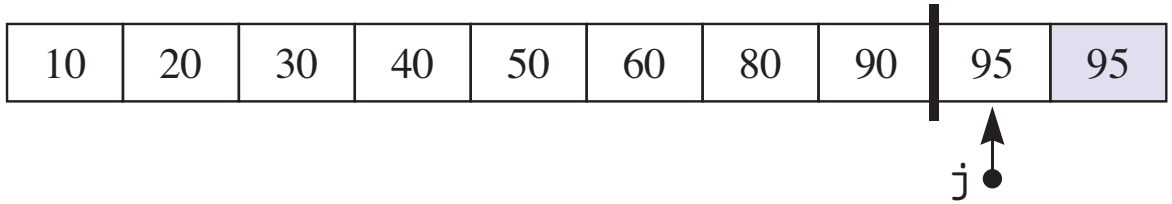
(h) Compare 70 and 80. Because 80 is greater than 70, set `a[j]` to 80 and decrement `j`.



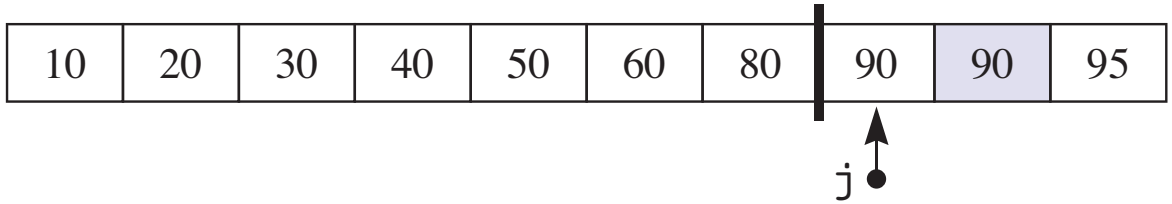
(e) `join(a, lo, mid, hi)`
Initialize `j` to `mid` and `key` to `a[mid]`, which is 70.



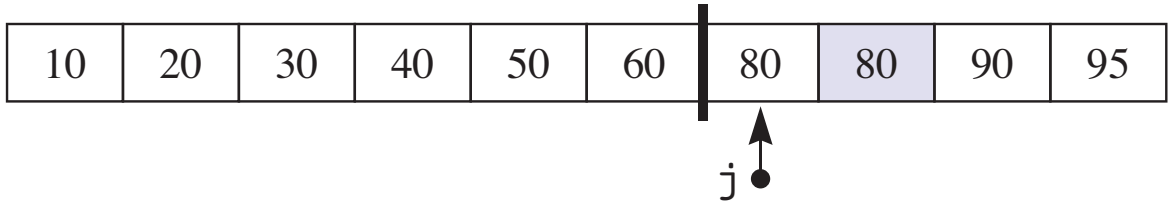
(f) Compare 70 and 95. Because 95 is greater than 70, set `a[j]` to 95 and decrement `j`.



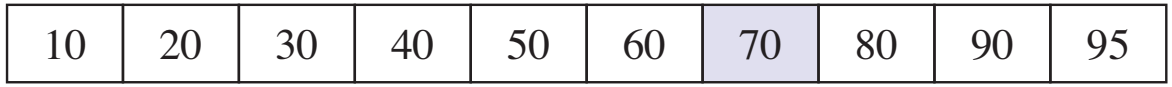
(g) Compare 70 and 90. Because 90 is greater than 70, set `a[j]` to 90 and decrement `j`.



(h) Compare 70 and 80. Because 80 is greater than 70, set `a[j]` to 80 and decrement `j`.



(i) Compare 70 and 60. Because 60 is not greater than 70, set `a[j]` to `key` and terminate the loop.



The performance of `InsertSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the recurrence?

The performance of `InsertSorter`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

What is the recurrence?

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Solution to the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

is

$$T(n) = \Theta(n^2)$$

Heap Sort

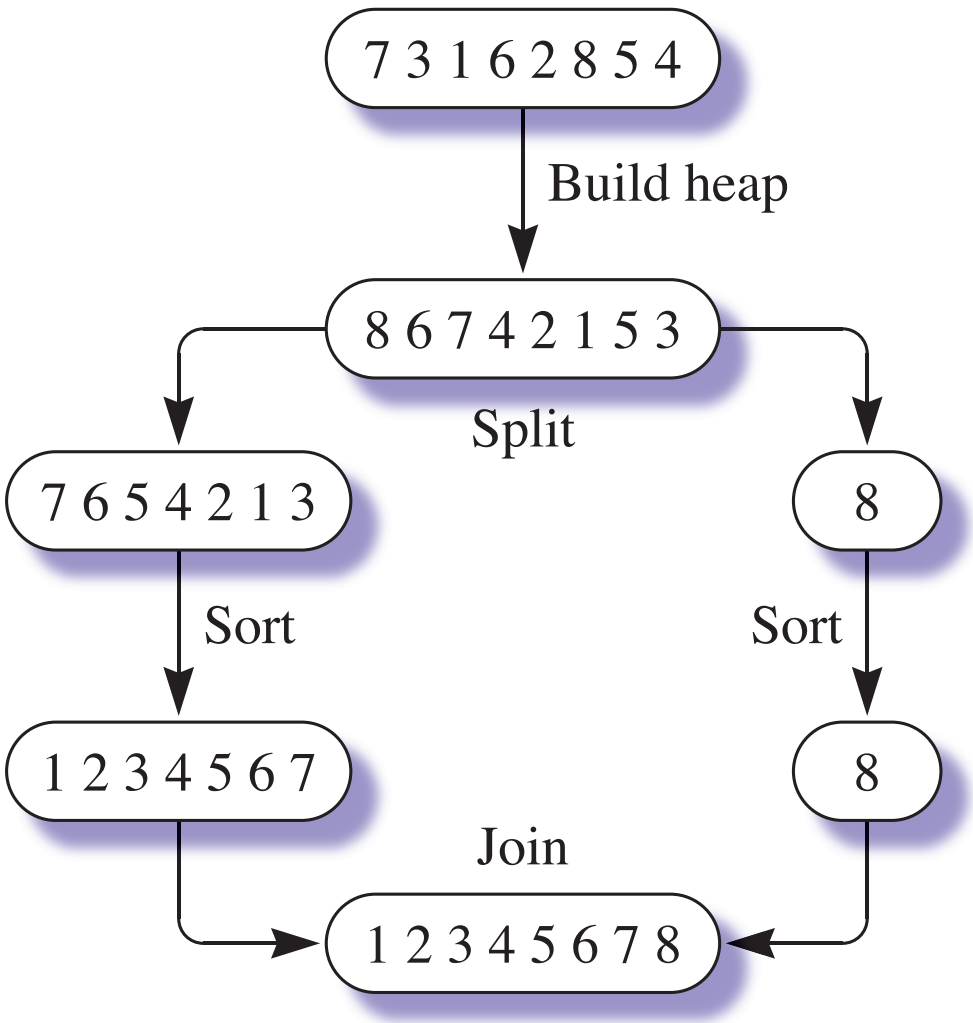
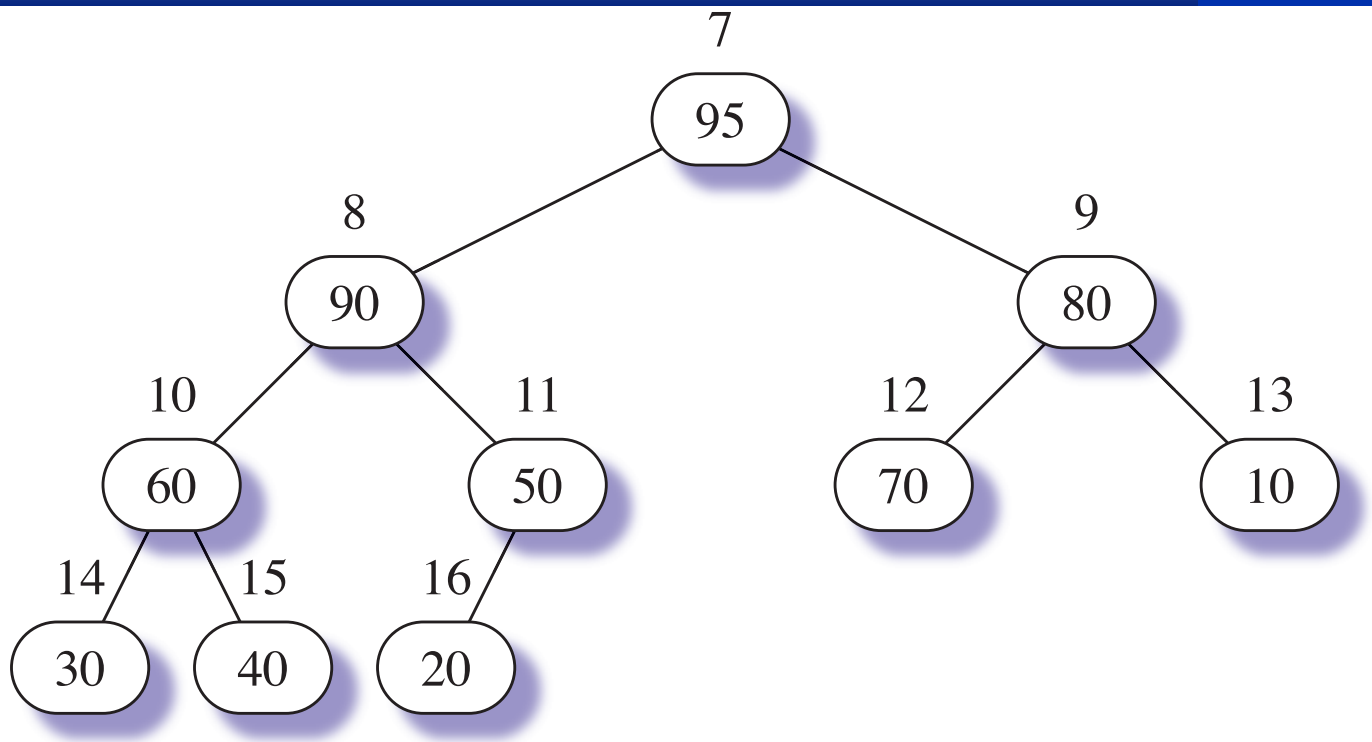
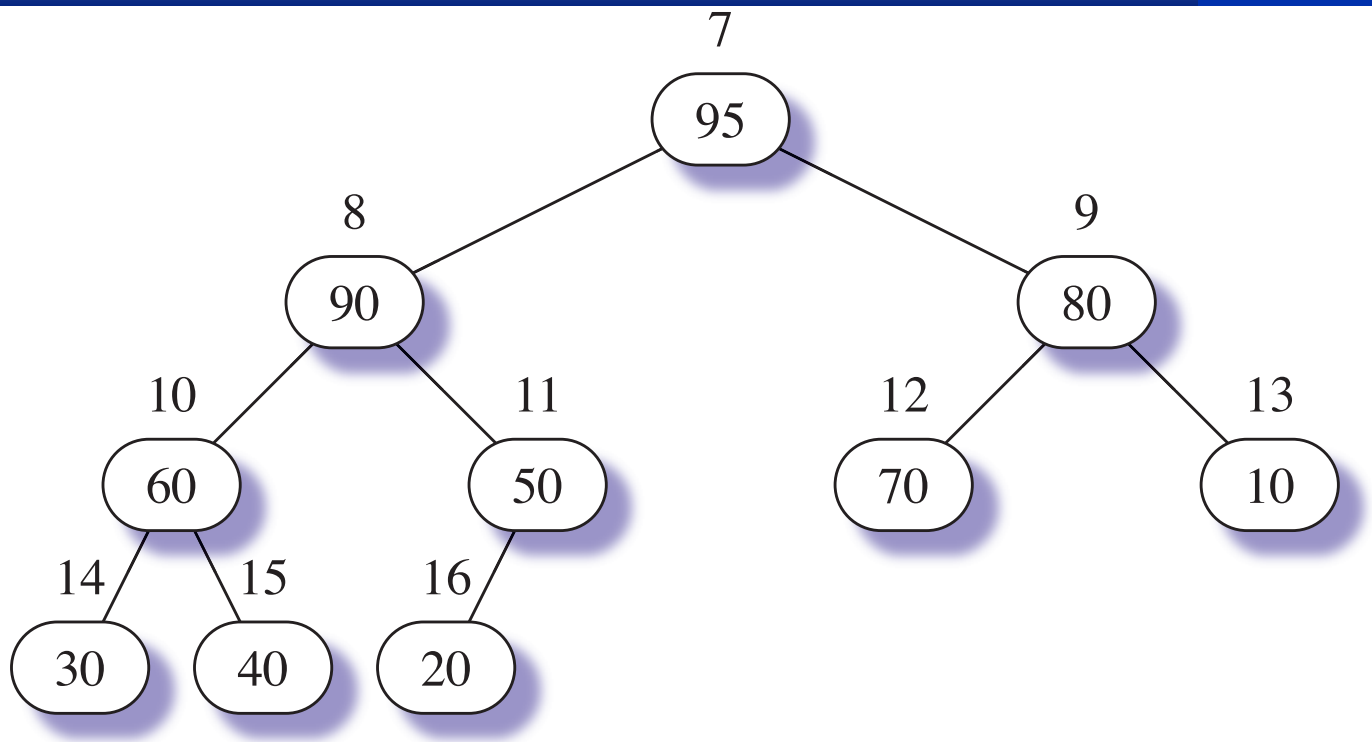

7 3 1 6 2 8 5 4
Build heap
8 6 7 4 2 1 5 3
Split
7 6 5 4 2 1 3
8
Sort
Sort
1 2 3 4 5 6 7
8
Join
1 2 3 4 5 6 7 8

Figure 4.23



[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
95	90	80	60	50	70	10	30	40	20

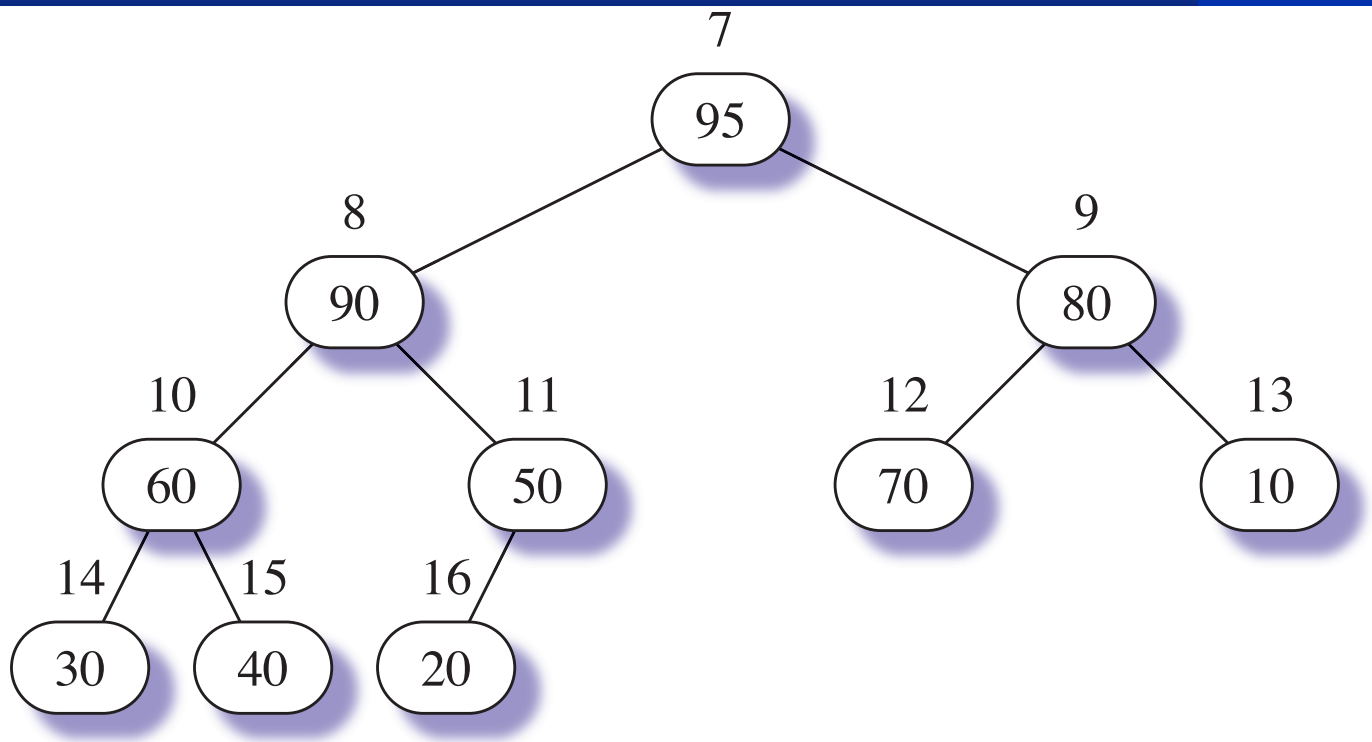
Figure 4.23



[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
95	90	80	60	50	70	10	30	40	20

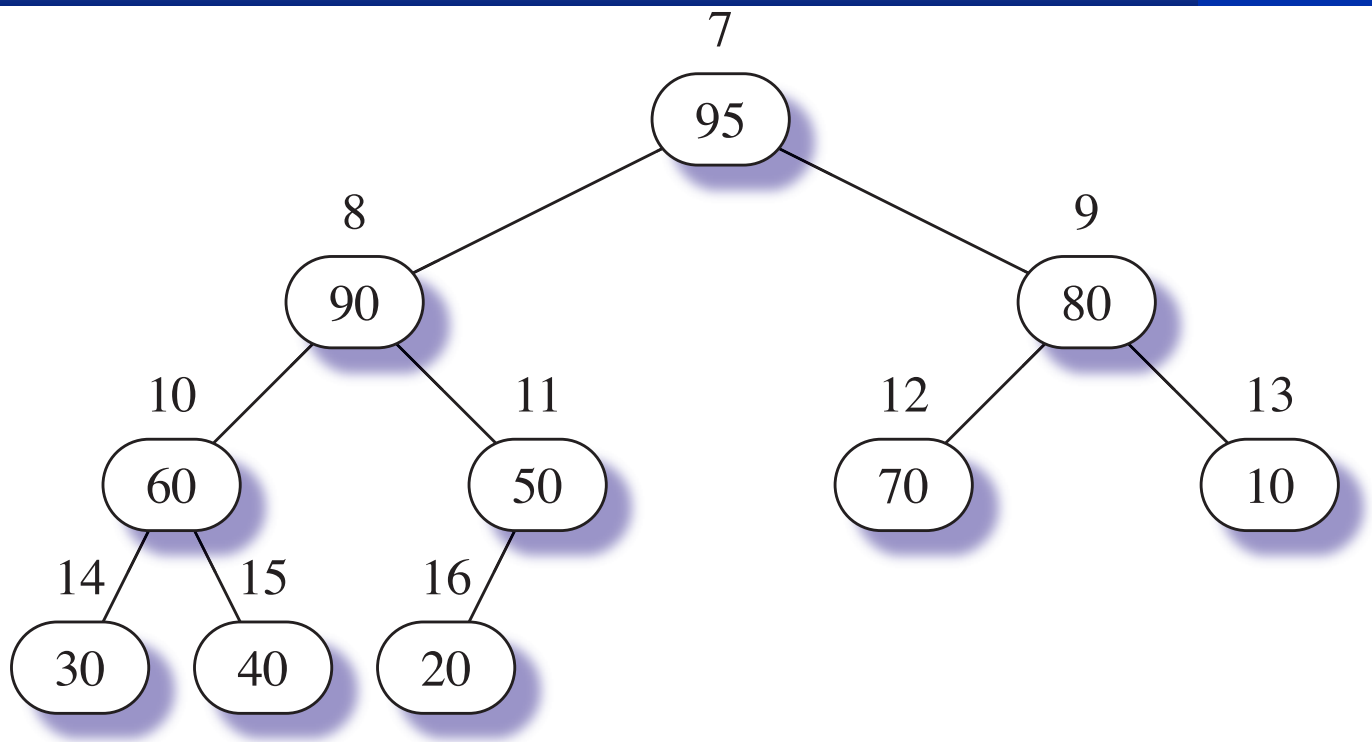
- The parent of the node at index i is at index $\lfloor (i+l-1)/2 \rfloor$.

Figure 4.23



[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
95	90	80	60	50	70	10	30	40	20

- The parent of the node at index i is at index $\lfloor (i+l-1)/2 \rfloor$.
- The left child of the node at index i is at index $2 \cdot i - l + 1$.

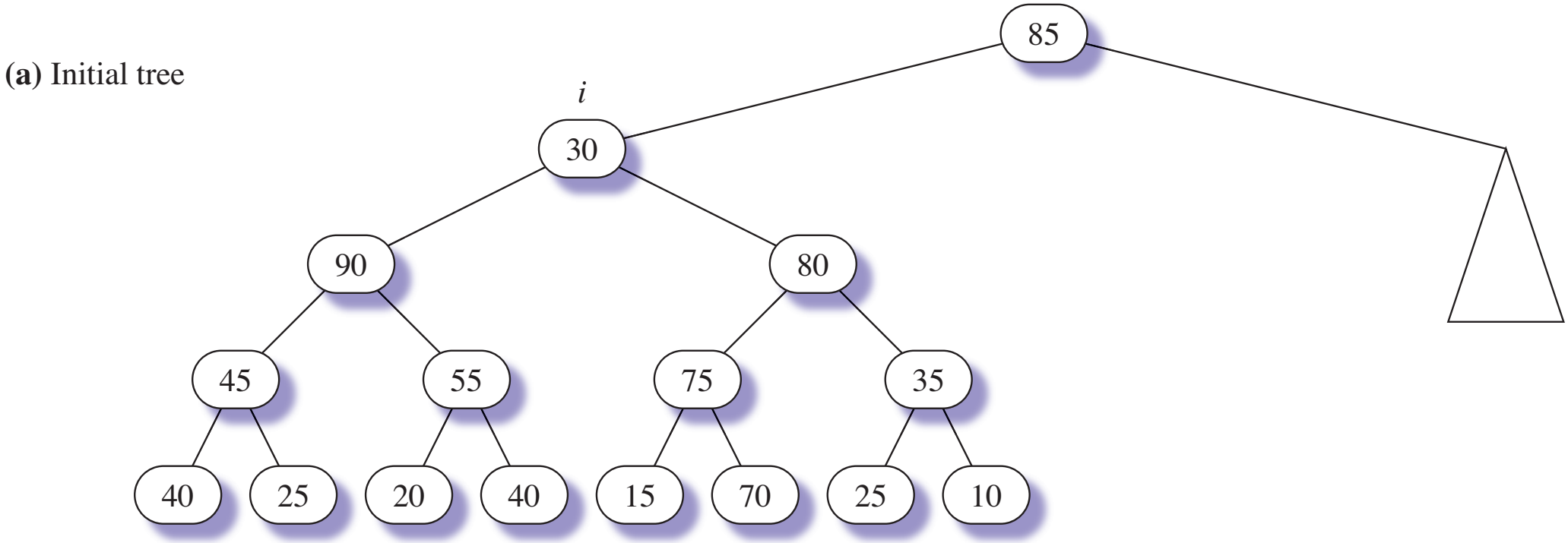


[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
95	90	80	60	50	70	10	30	40	20

- The parent of the node at index i is at index $\lfloor (i+l-1)/2 \rfloor$.
- The left child of the node at index i is at index $2 \cdot i - l + 1$.
- The largest index of a node with at least one child is $\lfloor (l+h-1)/2 \rfloor$.

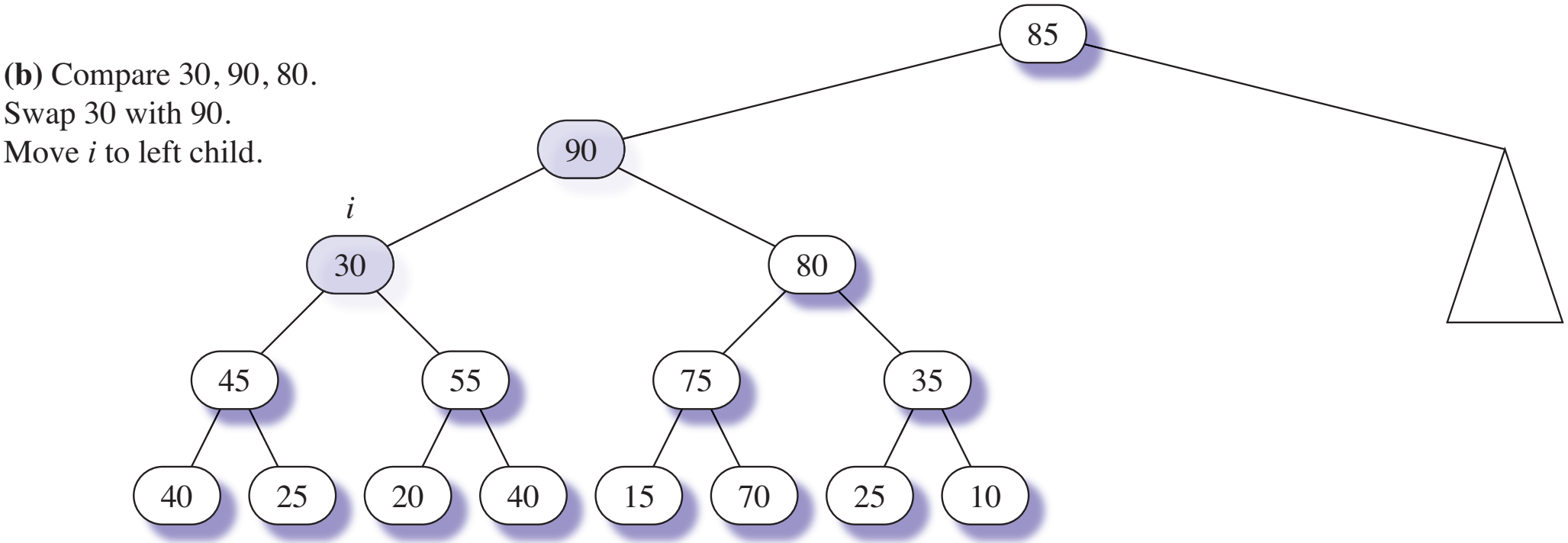
SiftDown()

Pre: Every node below i satisfies the maxheap property.
Post: Every node including i satisfies the maxheap property.



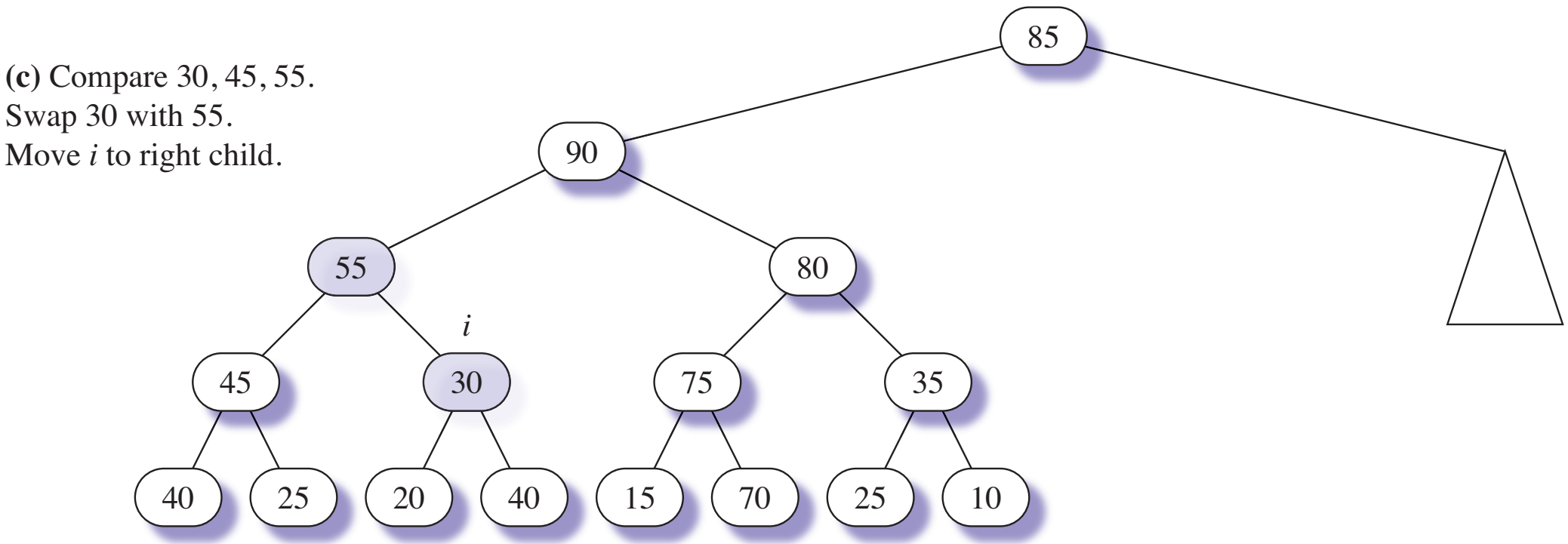
SiftDown()

Pre: Every node below i satisfies the maxheap property.
Post: Every node including i satisfies the maxheap property.



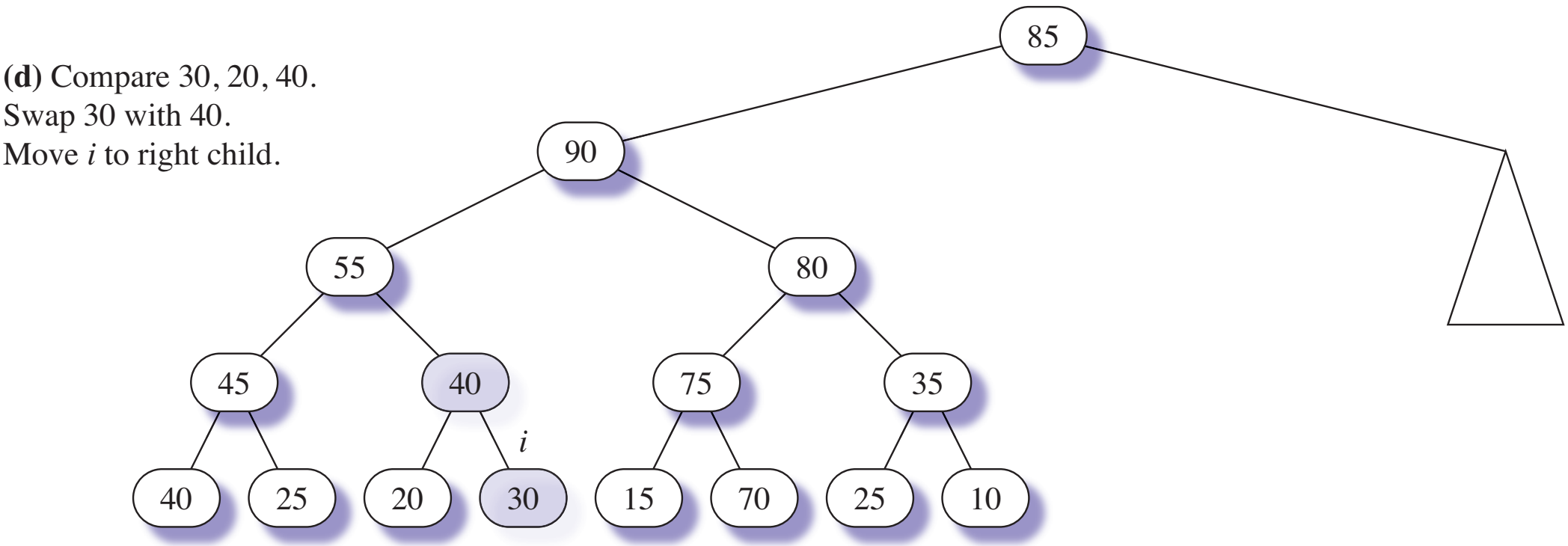
SiftDown()

Pre: Every node below i satisfies the maxheap property.
Post: Every node including i satisfies the maxheap property.



SiftDown()

Pre: Every node below i satisfies the maxheap property.
Post: Every node including i satisfies the maxheap property.



```
// ========= siftDown =========
template<class T>
void siftDown(ASeq<T> &a, int lo, int i, int hi) {
// Pre: maxHeap(a[i + 1..hi]).
// Pre: lo <= i <= hi.
// Post: maxHeap(a[i..hi]).
    int child = 2 * i - lo + 1; // Index of left child.
    if (child <= hi) {
        if (child < hi && a[child] < a[child + 1]) {
            child++;
        } // child is the index of the larger of the two children.
        if (a[i] < a[child]) {
            T temp = a[i];
            a[i] = a[child];
            a[child] = temp;
            siftDown(a, lo, child, hi);
        }
    }
}
```

The worst case performance of `siftDown`

```
if (child <= hi) {
    if (child < hi && a[child] < a[child + 1]) {
        child++;
    }  // child is the index of the larger of the two children.
    if (a[i] < a[child]) {
        T temp = a[i];
        a[i] = a[child];
        a[child] = temp;
        siftDown(a, lo, child, hi);
    }
}
```

The worst case performance of `siftDown`

```
if (child <= hi) {
    if (child < hi && a[child] < a[child + 1]) {
        child++;
    }  // child is the index of the larger of the two children.
    if (a[i] < a[child]) {
        T temp = a[i];
        a[i] = a[child];
        a[child] = temp;
        siftDown(a, lo, child, hi);
    }
}
```

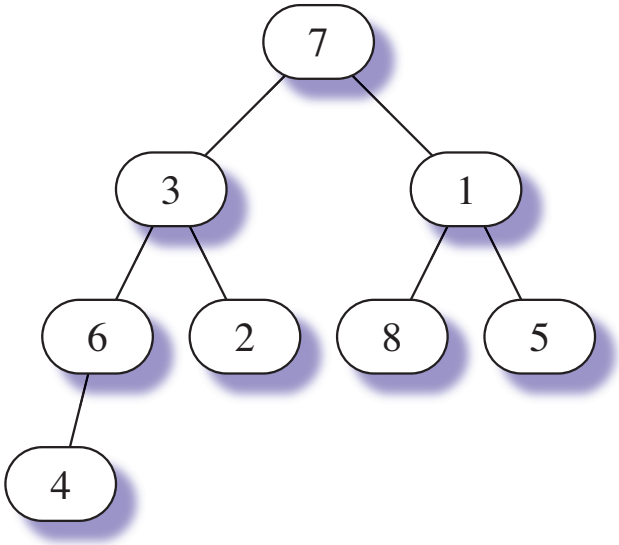
Height of heap with n elements

The worst case performance of `siftDown`

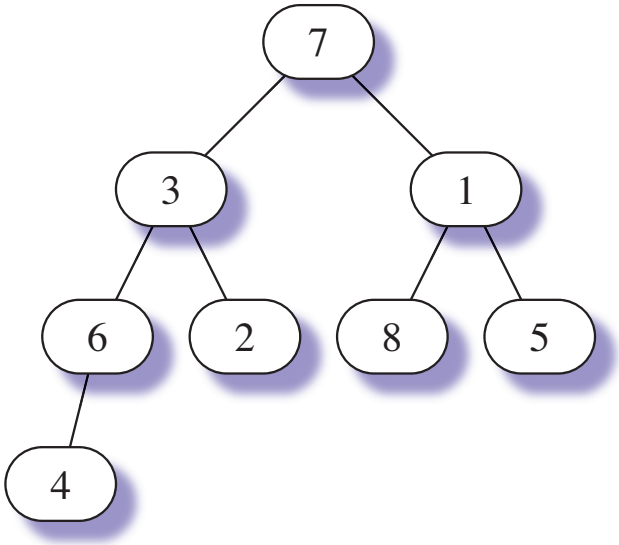
```
if (child <= hi) {
    if (child < hi && a[child] < a[child + 1]) {
        child++;
    }  // child is the index of the larger of the two children.
    if (a[i] < a[child]) {
        T temp = a[i];
        a[i] = a[child];
        a[child] = temp;
        siftDown(a, lo, child, hi);
    }
}
```

Height of heap with n elements

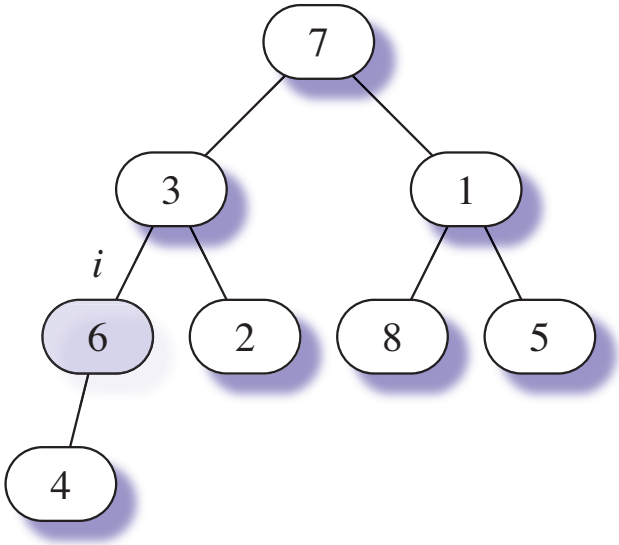
$$T(n) = \Theta(\lg n)$$



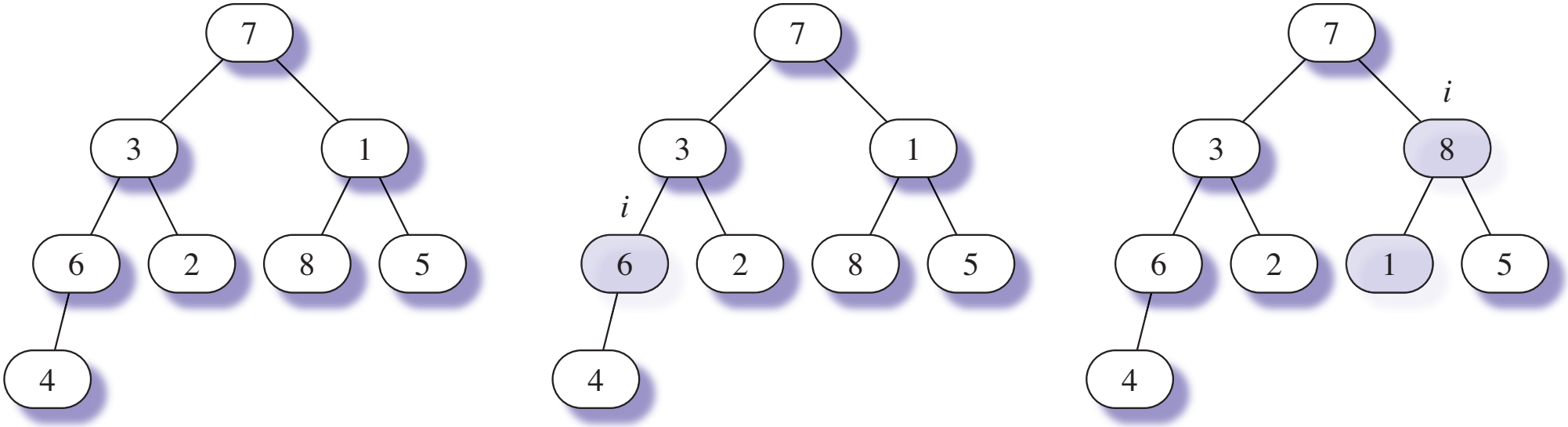
(a) The initial list is not a heap.



(a) The initial list is not a heap.



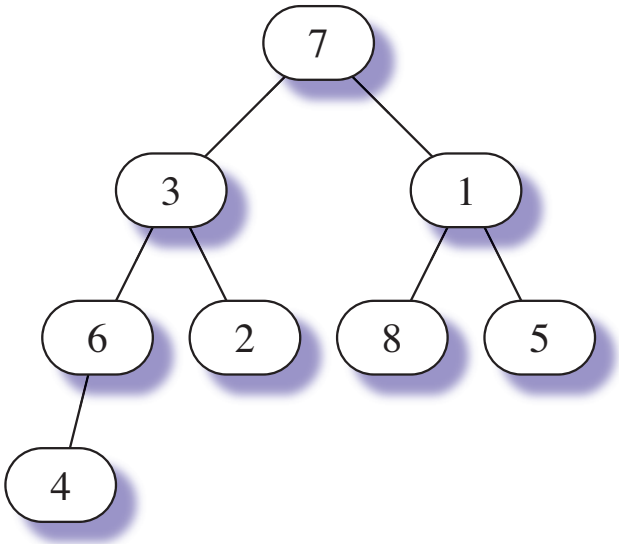
(b) Sift down 6.



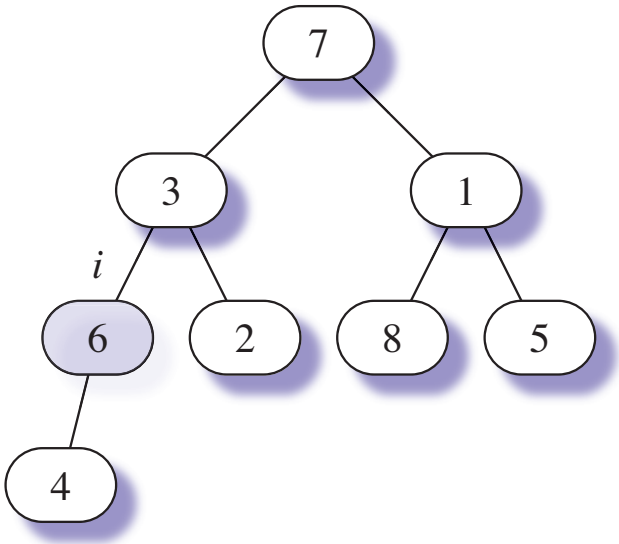
(a) The initial list is not a heap.

(b) Sift down 6.

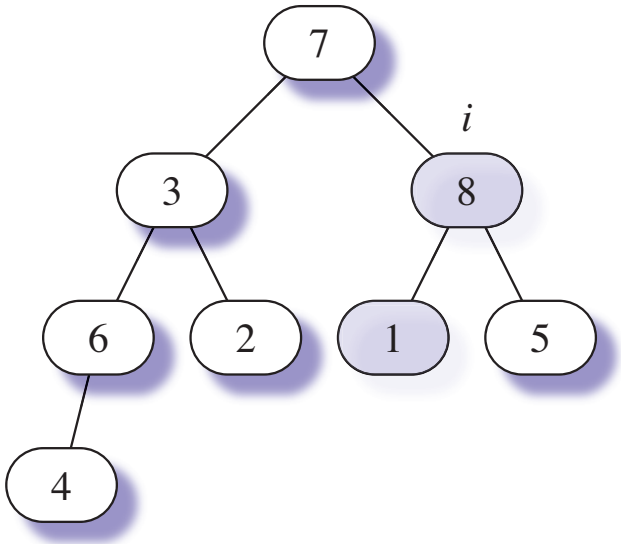
(c) Sift down 1.



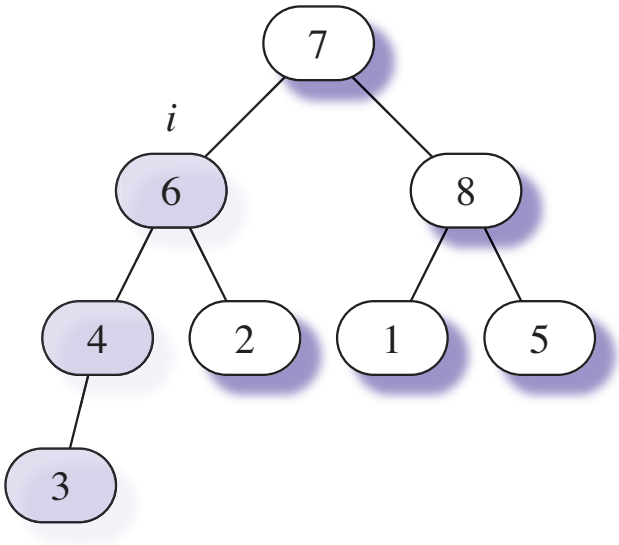
(a) The initial list is not a heap.



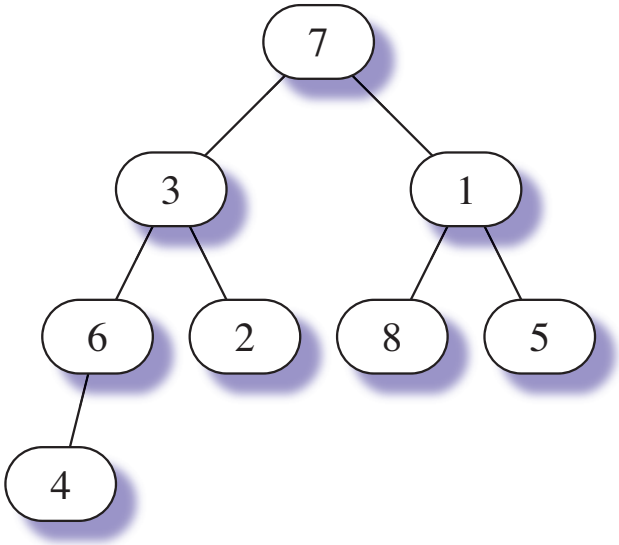
(b) Sift down 6.



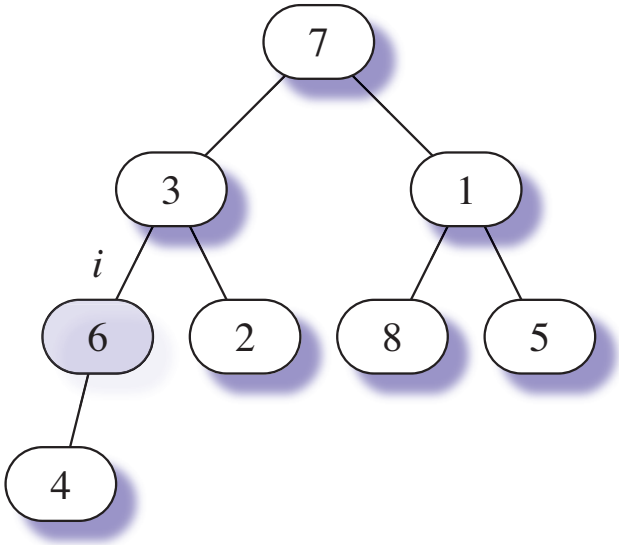
(c) Sift down 1.



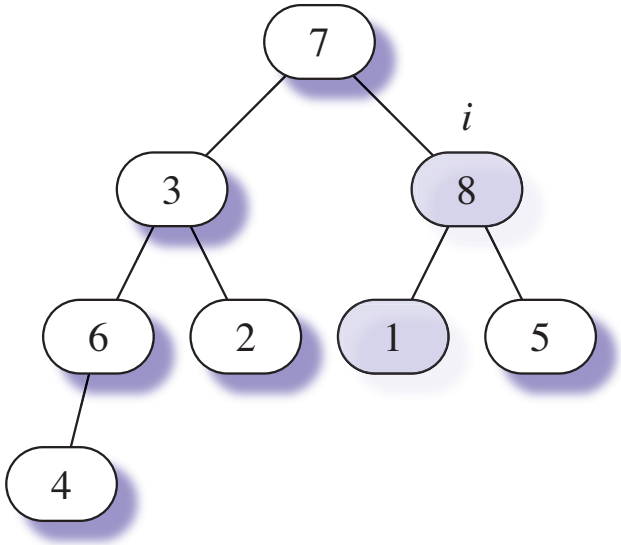
(d) Sift down 3.



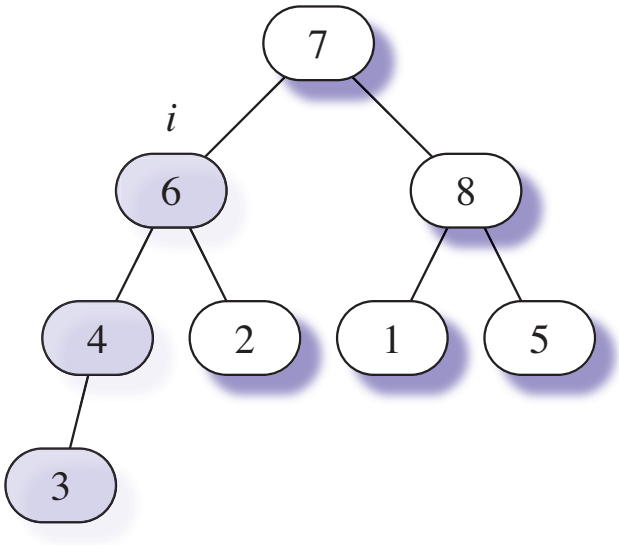
(a) The initial list is not a heap.



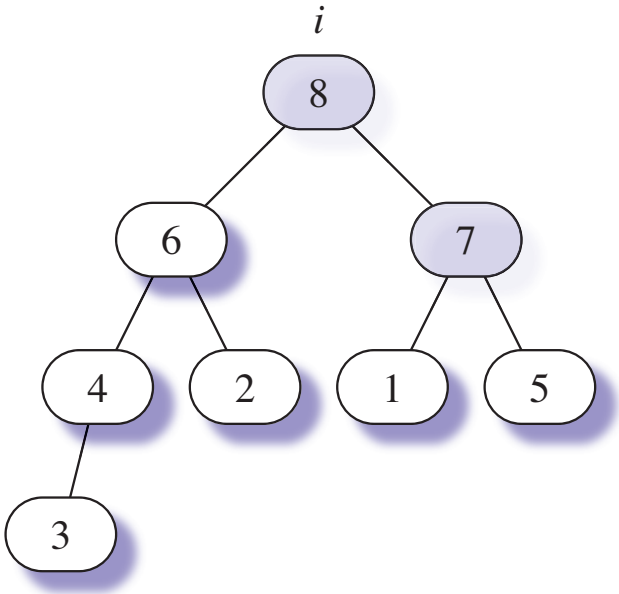
(b) Sift down 6.



(c) Sift down 1.



(d) Sift down 3.



(e) Sift down 7.

Figure 4.21

```
template<class T>
class HeapSorter : public ASorter<T> {
public:
    HeapSorter(ASeq<T> &a, int lo, int hi);
    // Constructor initializes a to a heap.
    // Post: maxHeap(a[lo..hi]).

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};

template<class T>
HeapSorter<T>::HeapSorter(ASeq<T> &a, int lo, int hi) {
// Post: maxHeap(a[lo..hi]).
    for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
        siftDown(a, lo, i, hi);
    }
}
```

The worst case performance to build initial heap

```
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

The worst case performance to build initial heap

```
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

Loop executes n/2 times.

The worst case performance to build initial heap

```
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

Loop executes n/2 times.

Each time, worst case lg n executions.

The worst case performance to build initial heap

```
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

Loop executes n/2 times.

Each time, worst case lg n executions.

$$T(n) = \Theta(n \lg n)$$

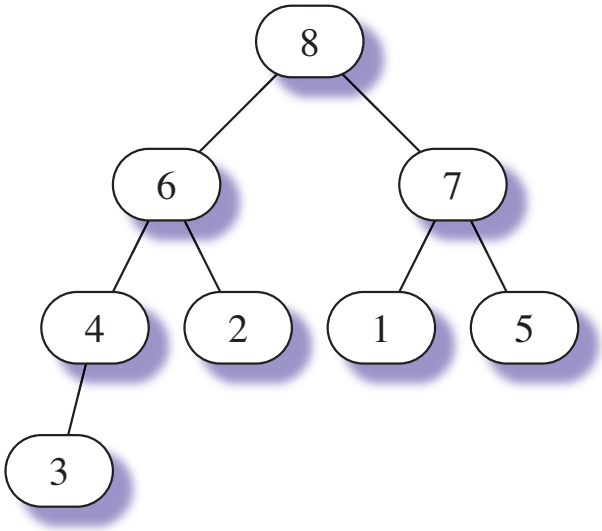
The worst case performance to build initial heap

```
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

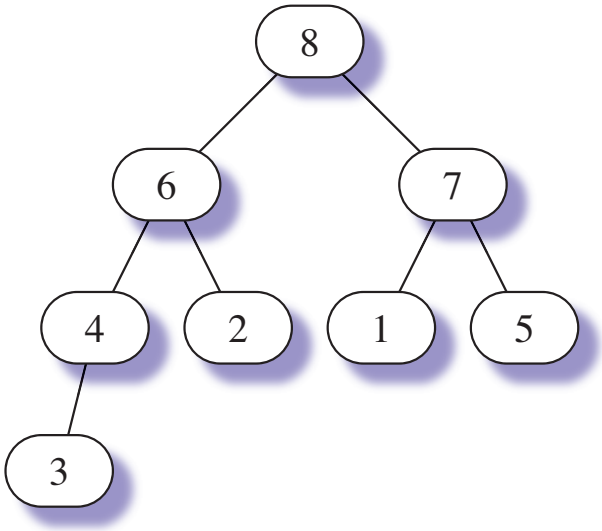
Can show actual performance is

$$T(n) = \Theta(n)$$

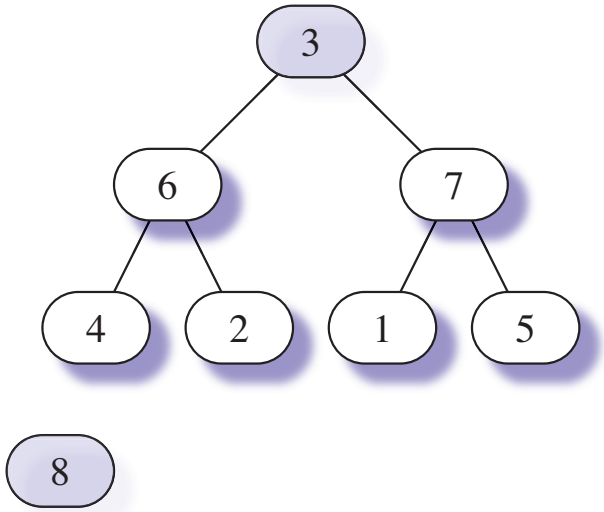
Beyond the scope of this book.



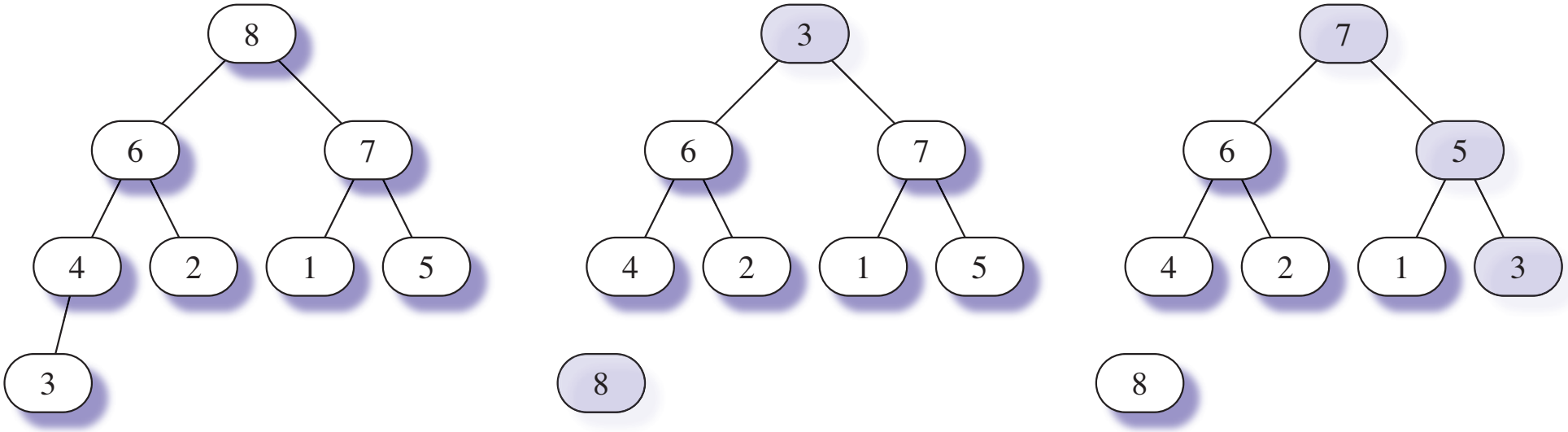
(a) The initial heap.



(a) The initial heap.



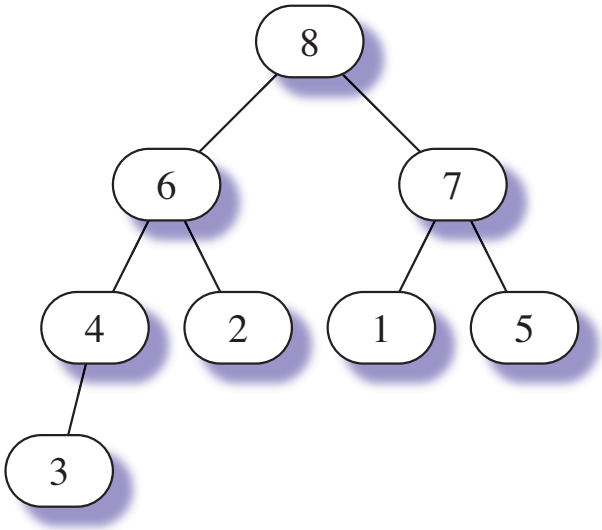
(b) Swap 8 and 3.



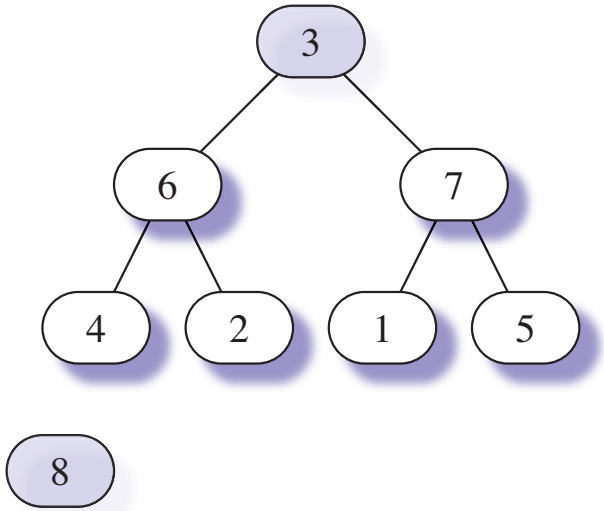
(a) The initial heap.

(b) Swap 8 and 3.

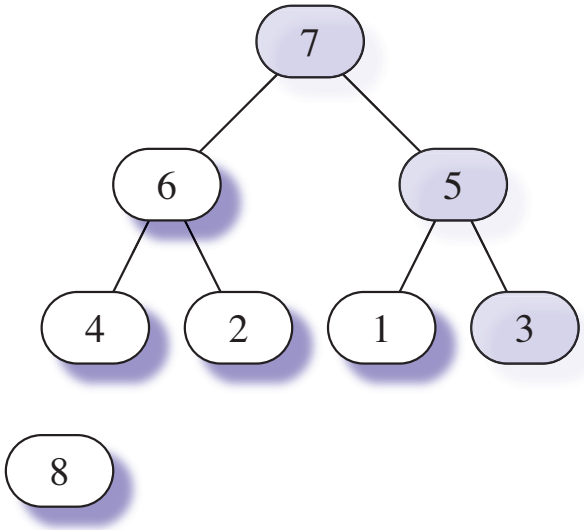
(c) Sift down 3.



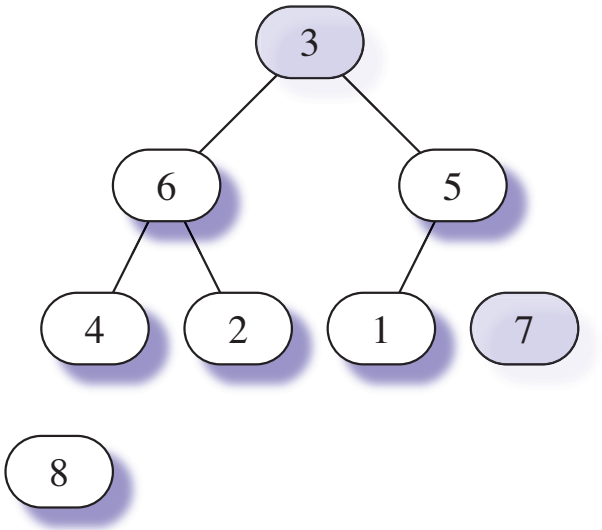
(a) The initial heap.



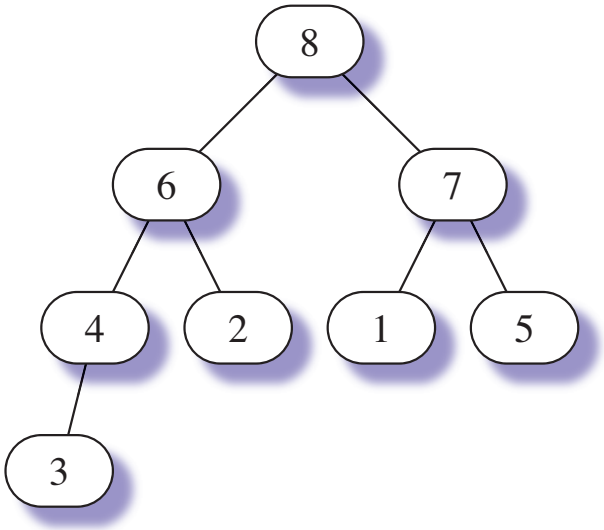
(b) Swap 8 and 3.



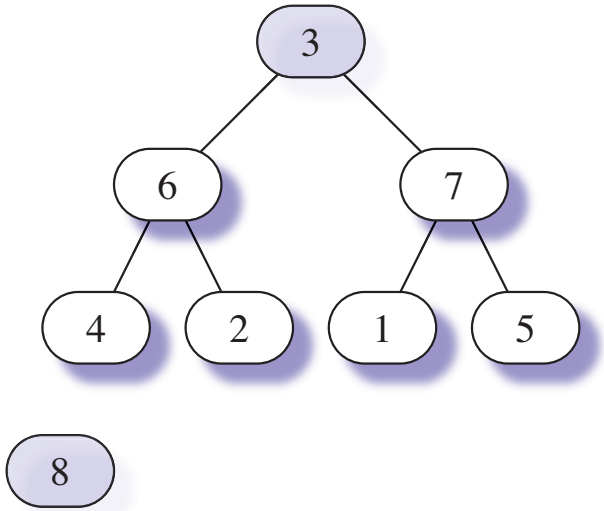
(c) Sift down 3.



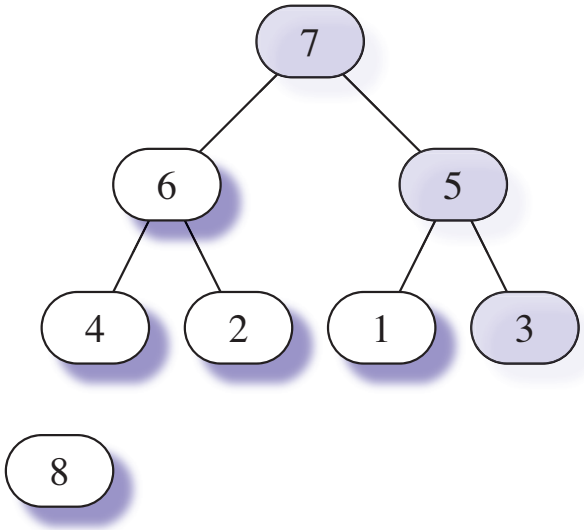
(d) Swap 7 and 3.



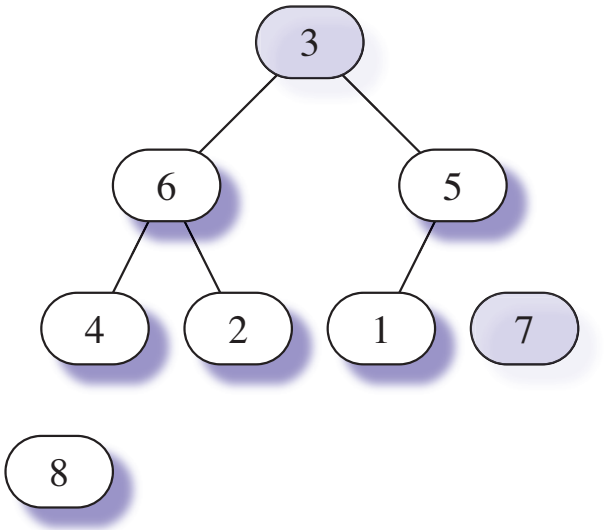
(a) The initial heap.



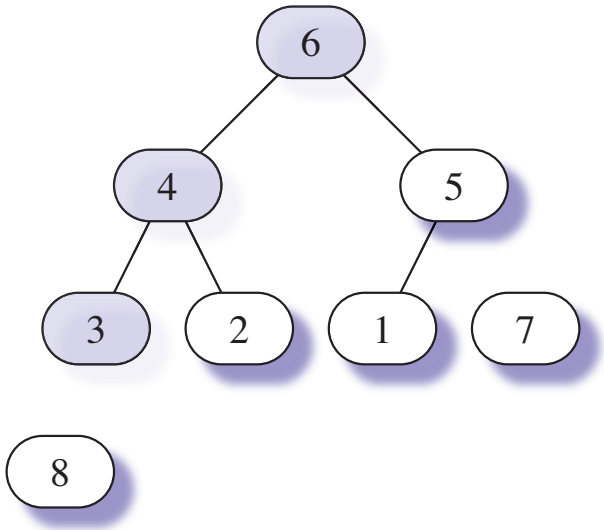
(b) Swap 8 and 3.



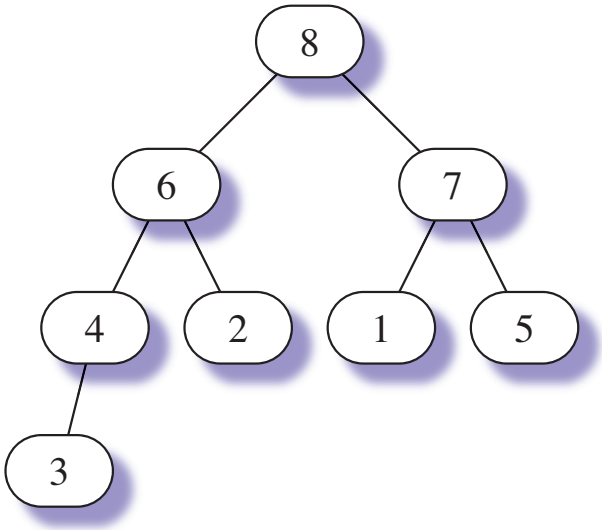
(c) Sift down 3.



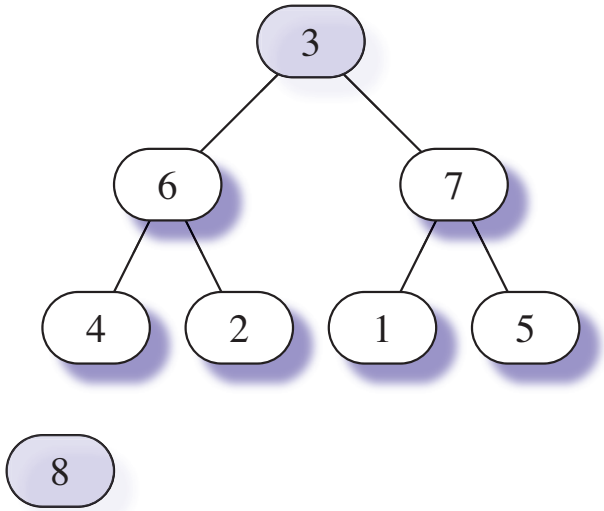
(d) Swap 7 and 3.



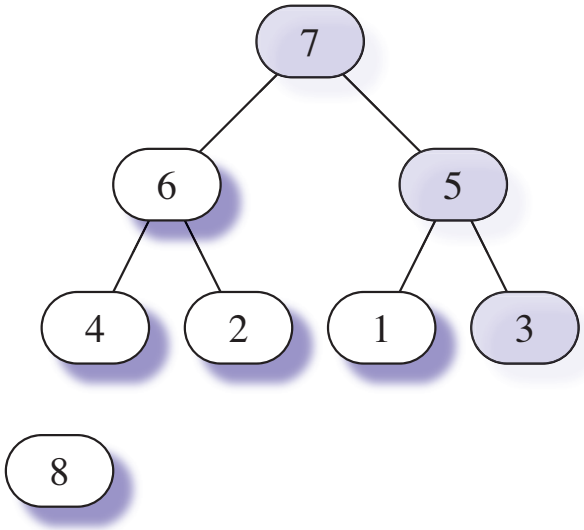
(e) Sift down 3.



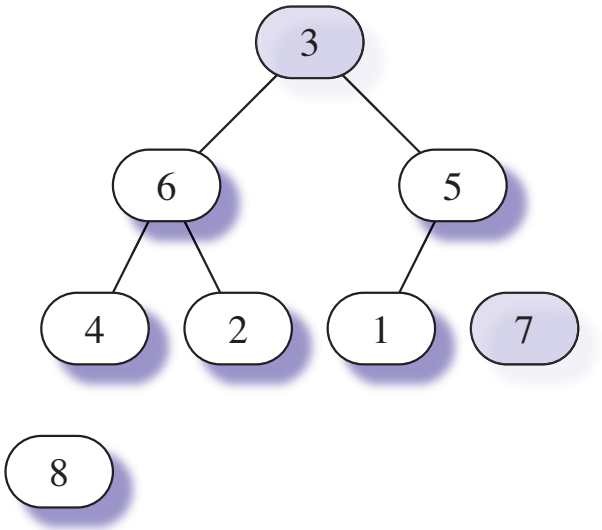
(a) The initial heap.



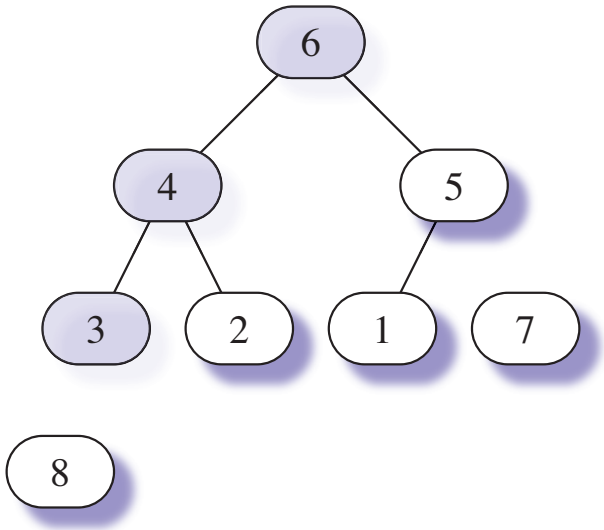
(b) Swap 8 and 3.



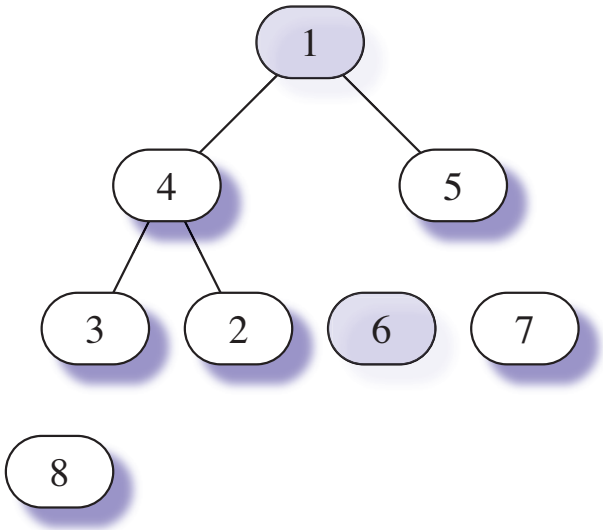
(c) Sift down 3.



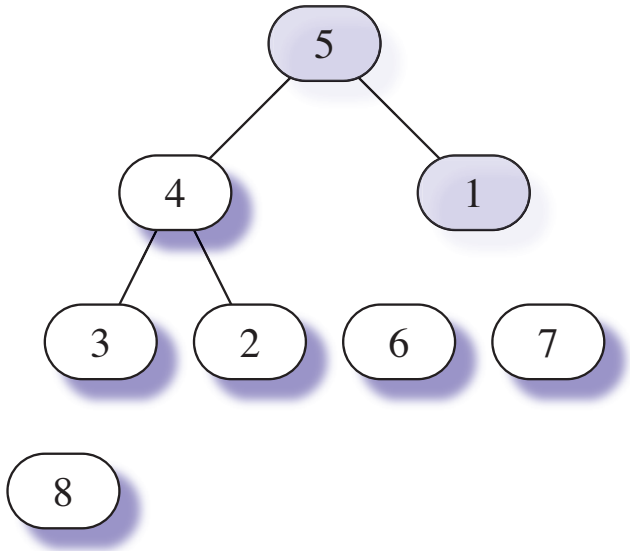
(d) Swap 7 and 3.



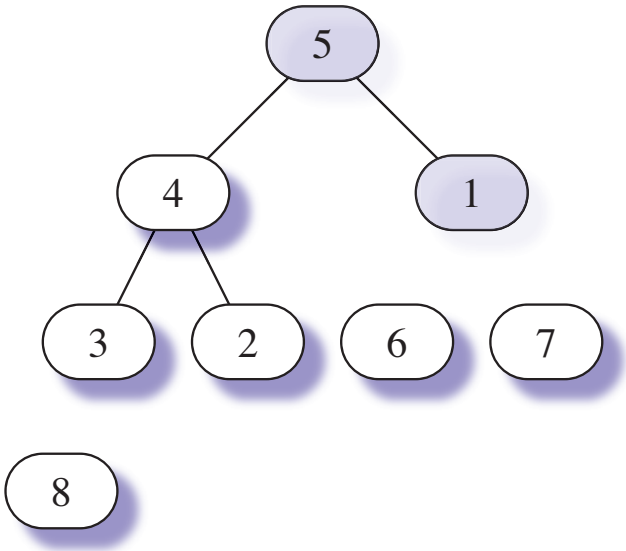
(e) Sift down 3.



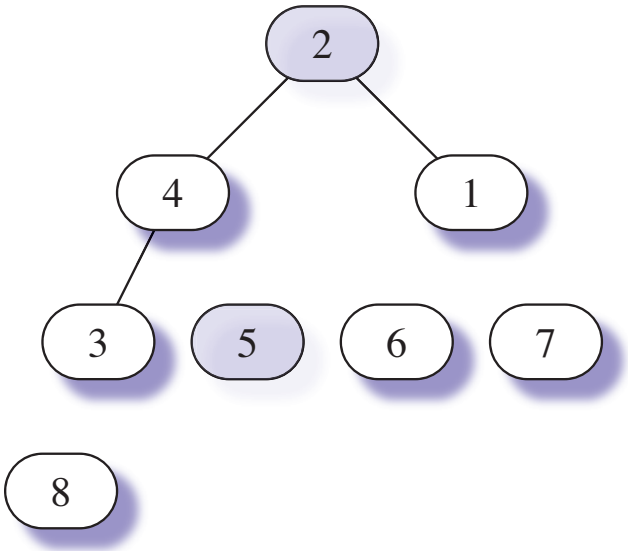
(f) Swap 6 and 1.



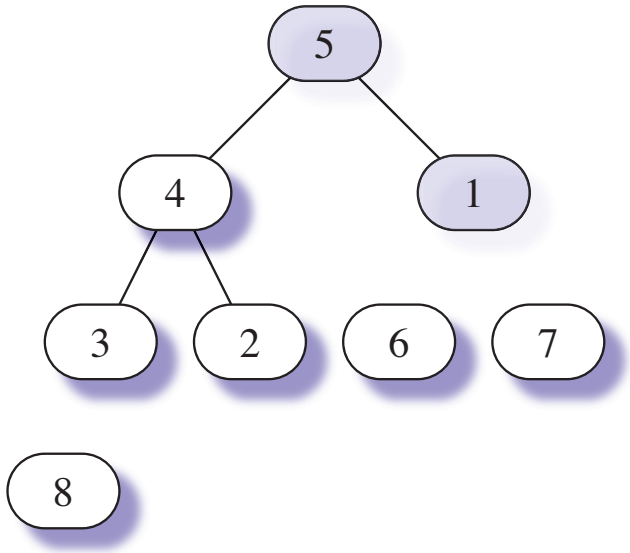
(g) Sift down 1.



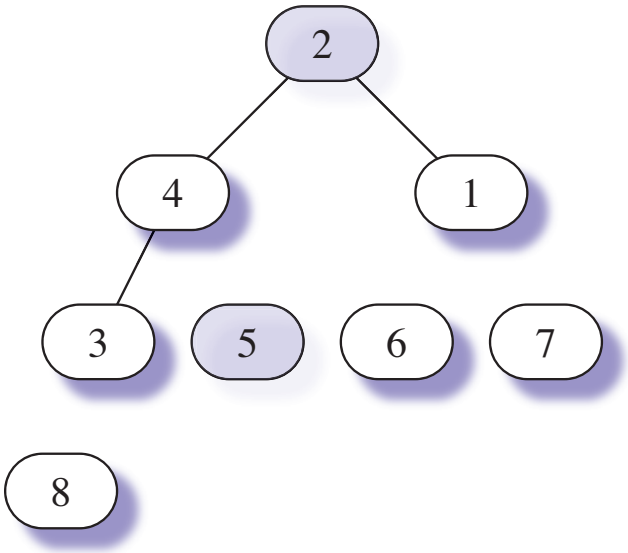
(g) Sift down 1.



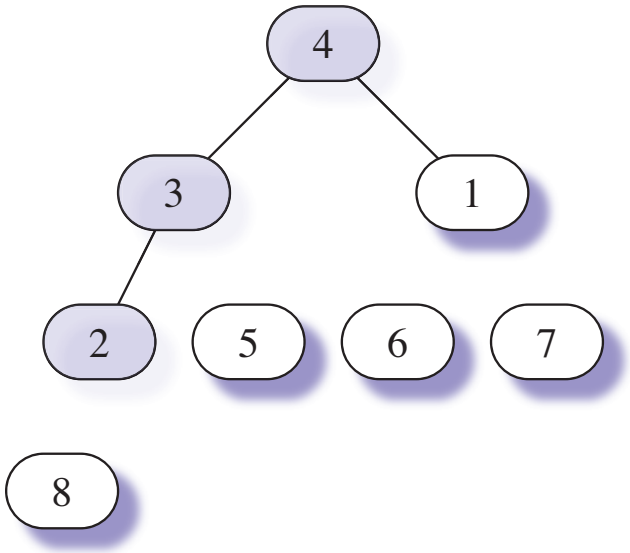
(h) Swap 5 and 2.



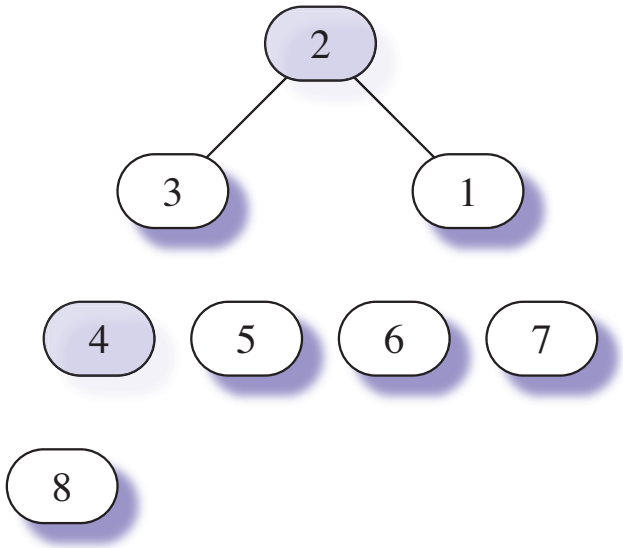
(g) Sift down 1.



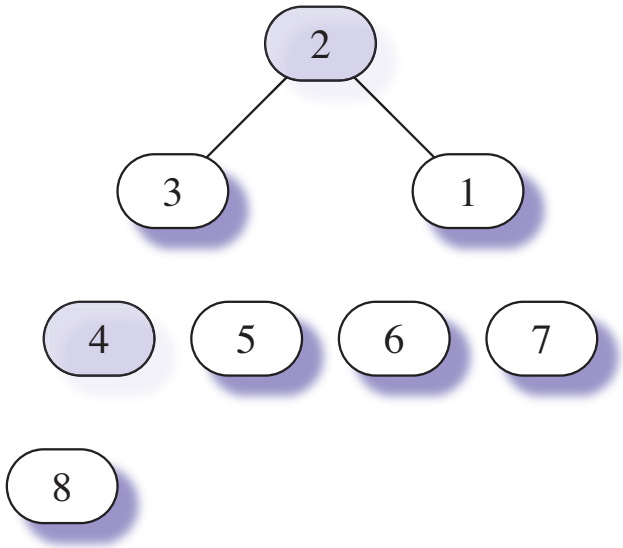
(h) Swap 5 and 2.



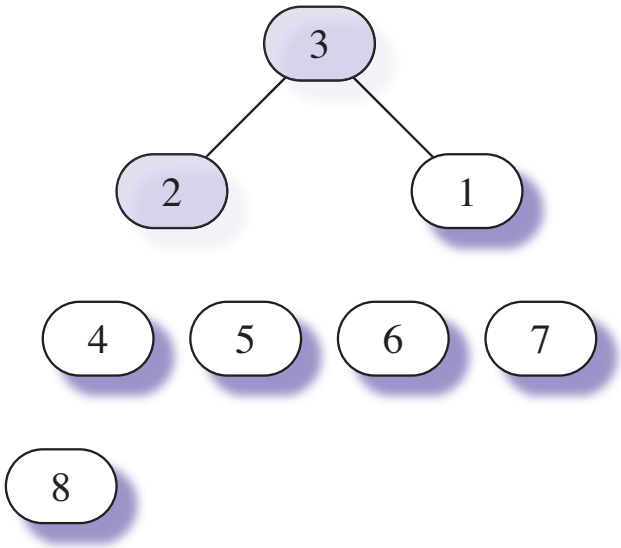
(i) Sift down 2.



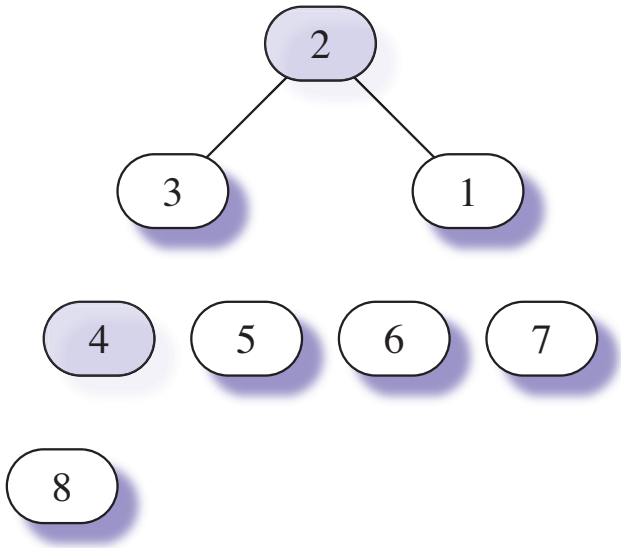
(j) Swap 4 and 2.



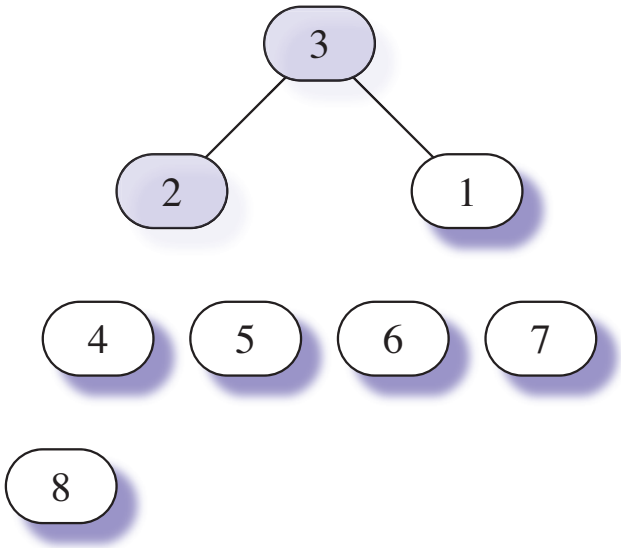
(j) Swap 4 and 2.



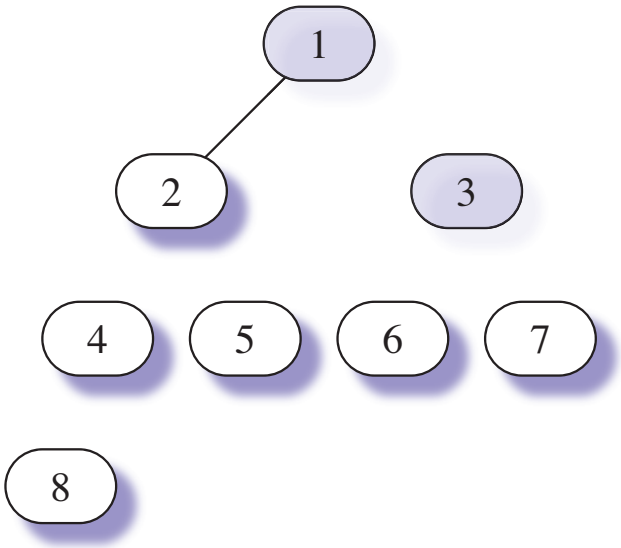
(k) Sift down 2.



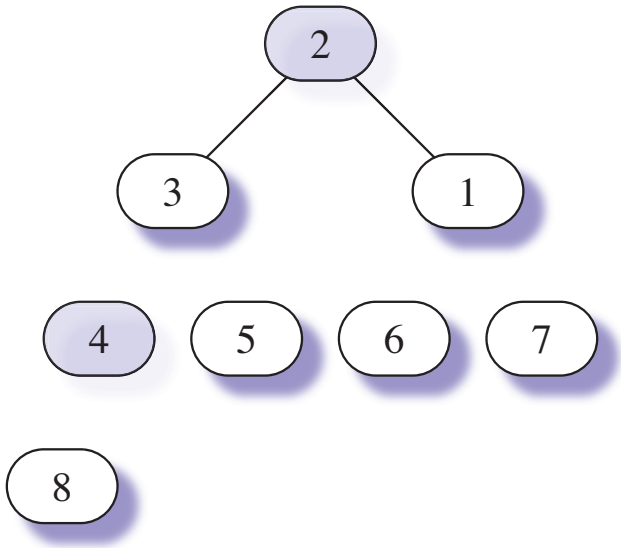
(j) Swap 4 and 2.



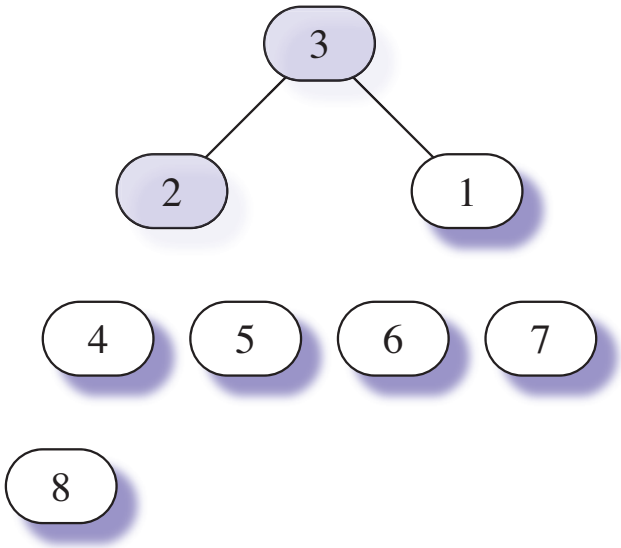
(k) Sift down 2.



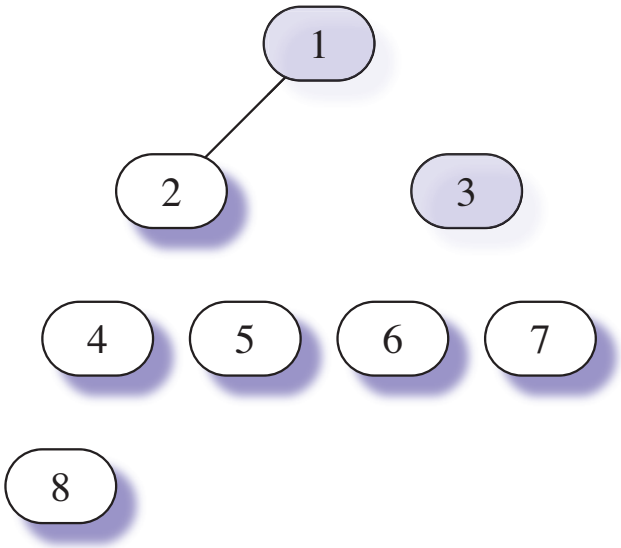
(l) Swap 3 and 1.



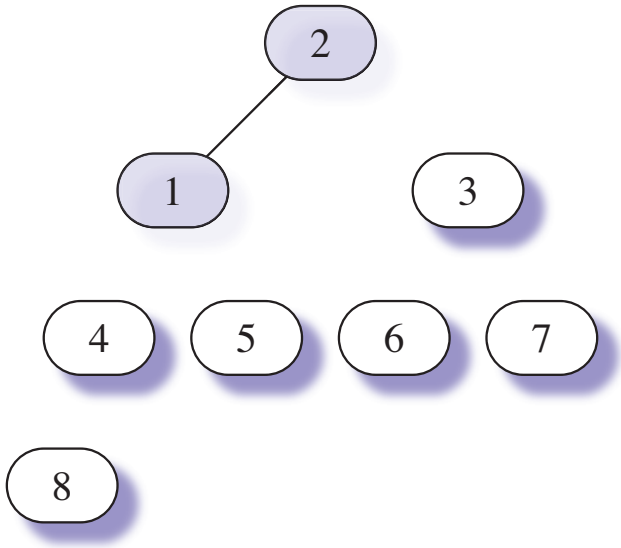
(j) Swap 4 and 2.



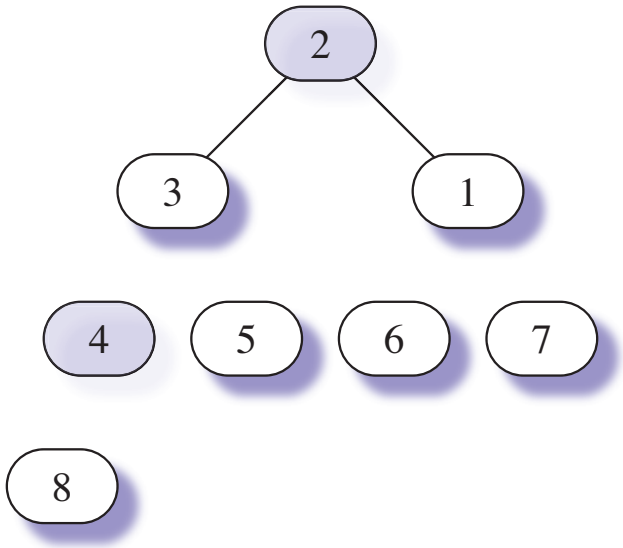
(k) Sift down 2.



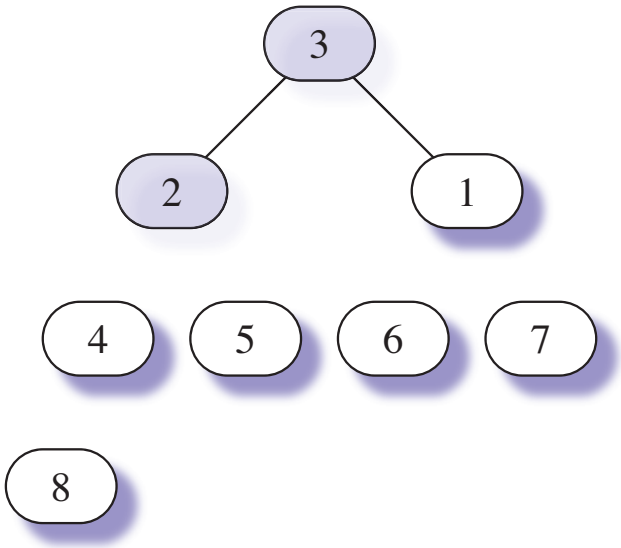
(l) Swap 3 and 1.



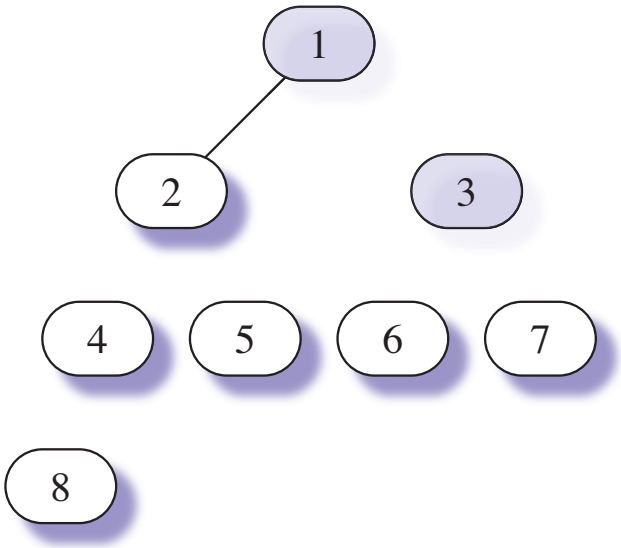
(m) Sift down 1.



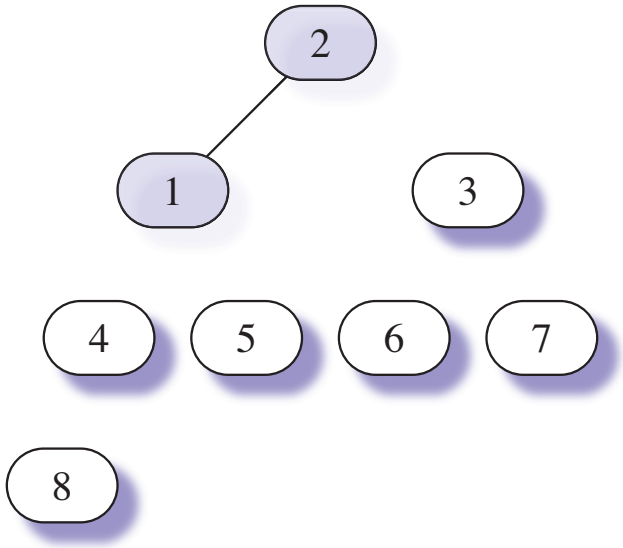
(j) Swap 4 and 2.



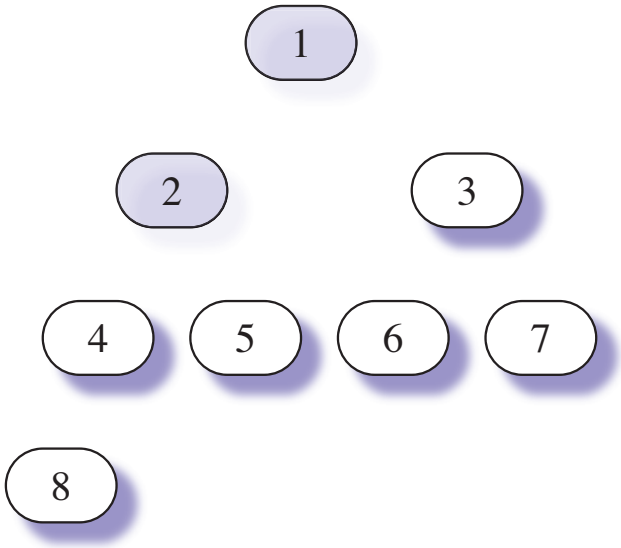
(k) Sift down 2.



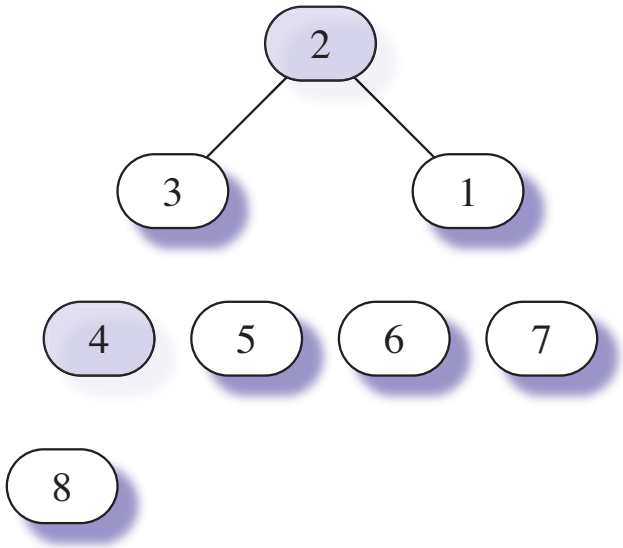
(l) Swap 3 and 1.



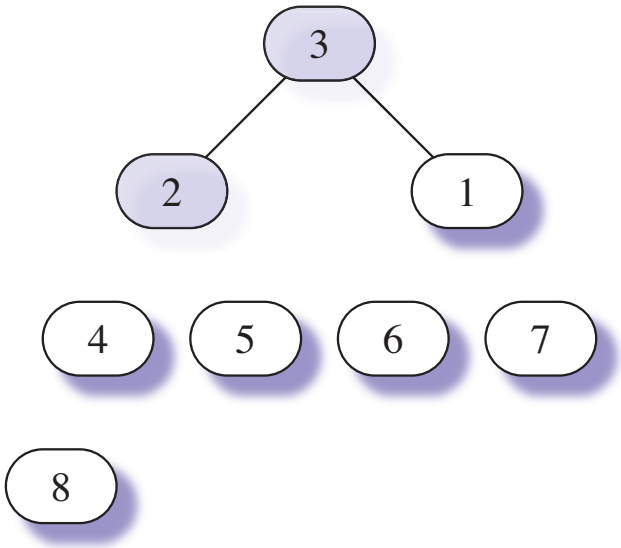
(m) Sift down 1.



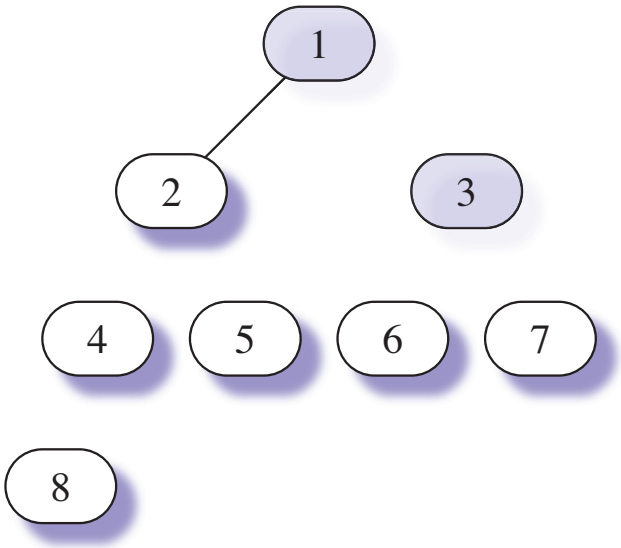
(n) Swap 2 and 1.



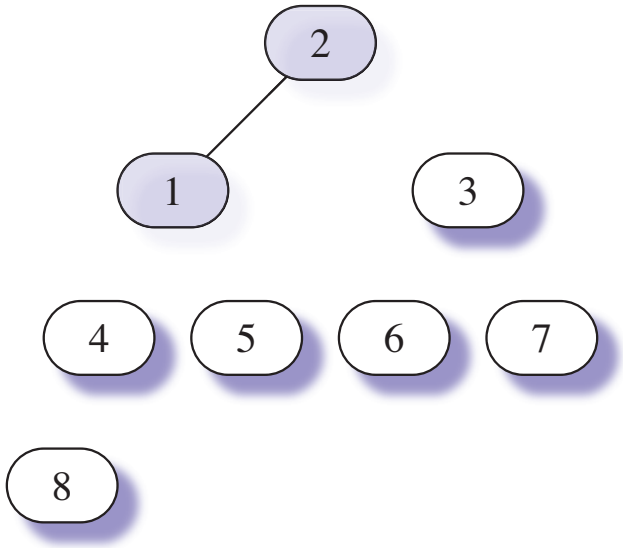
(j) Swap 4 and 2.



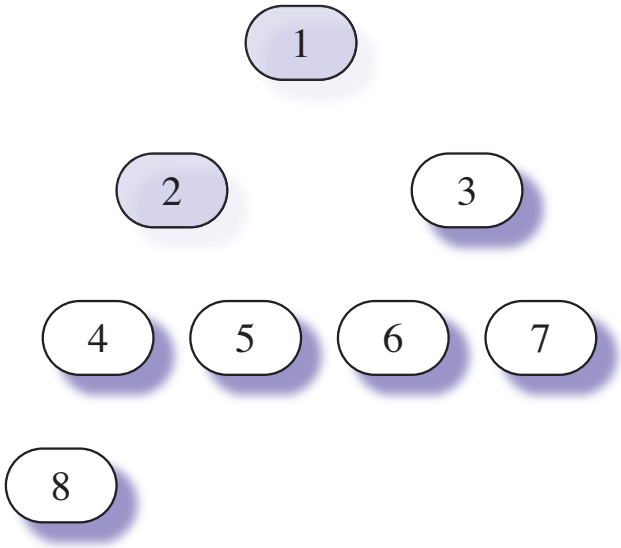
(k) Sift down 2.



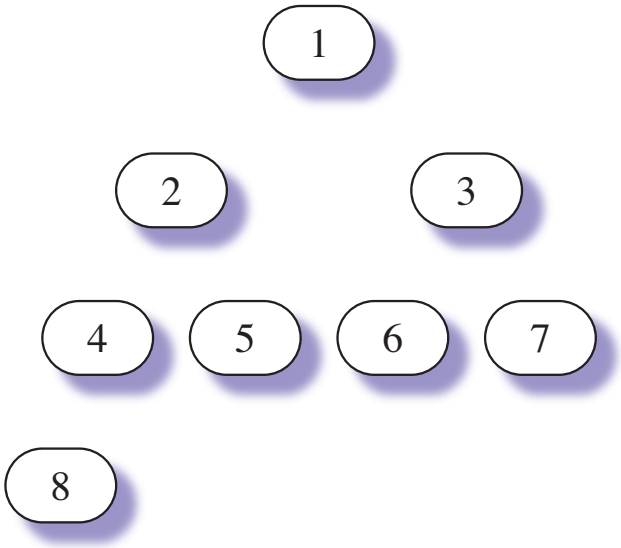
(l) Swap 3 and 1.



(m) Sift down 1.



(n) Swap 2 and 1.



(o) Sift down 1.

```
template<class T>
void HeapSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
// Pre: maxHeap(a[lo..hi]).
// Post: maxHeap(a[lo..hi - 1]).
// Post: a[hi] == old a[lo] && mid == hi.
    T temp = a[hi];
    a[hi] = a[lo];
    a[lo] = temp;
    siftDown(a, lo, lo, hi - 1);
    mid = hi;
}

template<class T>
void HeapSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
```

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

Second `sort()` does no work.

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

Second `sort()` does no work.

`join()` does no work.

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

What is the time for split?

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

What is the time for split?

$$\Theta(\lg n)$$

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

What is the time for split?

$$\Theta(\lg n)$$

How many times does it execute?

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

What is the time for split?

$$\Theta(\lg n)$$

How many times does it execute?

$$\Theta(n)$$

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

What is the time for split?

$$\Theta(\lg n)$$

How many times does it execute?

$$\Theta(n)$$

Time for `sort()`

The performance of `HeapSorter::sort()`

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

What is the time for split?

$$\Theta(\lg n)$$

How many times does it execute?

$$\Theta(n)$$

Time for `sort()`

$$\Theta(n \lg n)$$

The performance of `HeapSorter`

Time to build initial heap + time to sort

The performance of `HeapSorter`

Time to build initial heap + time to sort

$$\Theta(n \lg n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n)$$

The performance of `HeapSorter`

Time to build initial heap + time to sort

$$\Theta(n \lg n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n)$$

Assuming tighter bound to build initial heap

The performance of `HeapSorter`

Time to build initial heap + time to sort

$$\underline{\Theta(n \lg n)} + \Theta(n) \cdot \Theta(\lg n) = \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n)$$

Assuming tighter bound to build initial heap

$$\underline{\Theta(n)} + \Theta(n) \cdot \Theta(\lg n) = \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$$