Chapter 10. Elementary Data Structures

10.1-1.
$\{4\}$, $\{4, 1\}$, $\{4, 1, 3\}$, $\{4, 1\}$, $\{4, 1, 8\}$, $\{4, 1\}$.

10.1-2.
Align the first stack from the left and the second stack from the right. Overflow happens only if an object is pushed when two stack pointers are adjacent.

10.1-3.
$\{4\}$, $\{4, 1\}$, $\{4, 1, 3\}$, $\{1, 3\}$, $\{1, 3, 8\}$, $\{3, 8\}$.

10.1-4.

Algorithm 1: ENQUEUE(Q, x)

```
if Q.head == Q.tail +1 or (Q.head == 1 and Q.tail == Q.length)
 then
    error "overflow"
else
    Q[Q.tail] = x;
    if Q.tail == Q.length then
        Q.tail = 1;
    else
        Q.tail += 1;
    end
end
```

Algorithm 2: DEQUEUE(Q, x)

```
if $Q.head == Q.tail$ then
    error "underflow"
else
    x = Q[Q.head];
    if $Q.head == Q.length$ then
        Q.head = 1;
    else
        Q.head += 1;
    end
    return x;
end
```

10.1-5.

Algorithm 3: HEAD-ENQUEUE(Q, x)

```
if $Q.head == Q.tail + 1$ or ($Q.head == 1$ and $Q.tail == Q.length$)
  then
    error "overflow"
else
    if $Q.head == 1$ then
        Q.head = Q.length;
    else
        Q.head -= 1;
    end
    Q[Q.head] = x;
end
```

Algorithm 4: TAIL-ENQUEUE(Q, x)

```
if Q.head == Q.tail +1 or (Q.head == 1 and Q.tail == Q.length)
 then
    error "overflow"
else
    Q[Q.tail] = x;
    if Q.tail == Q.length then
        Q.tail = 1;
    else
        Q.tail += 1;
    end
end
```

Algorithm 5: HEAD-DEQUEUE(Q, x)

```
if Q.head == Q.tail then
    error "underflow"
else
    x = Q[Q.head];
    if Q.head == Q.length then
        Q.head = 1;
    else
        Q.head += 1;
    end
    return x;
end
```

Algorithm 6: TAIL-DEQUEUE(Q, x)

```
if Q.head == Q.tail then
    error "underflow"
else
    if Q.tail == 1 then
        Q.tail = Q.length;
    else
        Q.tail -= 1;
    end
    x = Q[Q.tail];
    return x;
end
```

10.1-6.
Let the two stacks be A and B. ENQUEUE pushes elements to A, DEQUEUE pops elements from B. If B is empty, whole elements in A is popped and pushed into B.
ENQUEUE is $\Theta(1)$, and DEQUEUE is amortized $\Theta(1)$.

10.1-7.
Let the two queues be A and B. PUSH enqueues elements to A. POP dequeues all but one element from A, and enqueues them to B, and dequeue the last element in A. After POP, the two queues should be swapped.
PUSH is $\Theta(1)$, and POP is $\Theta(n)$.

10.2-1.
INSERT can be implemented by prepending the element to the head of the list.
DELETE can be implemented by copying the value from the successor to the element you want to delete, and then delete the successor.

10.2-2.
PUSH can be implemented by prepending the element to the head of the list.
POP can be implemented by removing the head of the list.

10.2-3.
ENQUEUE can be implemented by appending the element to the tail of the list. We have to keep track of the tail.
DEQUEUE can be implemented by removing the head of the list.
10.2-4.

Algorithm 7: LIST-SEARCH'(L, k)

```
x = L.nil.next;
L.nil.key = k;
while $x.key \neq k$ do
    x = x.next;
end
return x;
```

10.2-5.

Algorithm 8: INSERT(D, k)

```
Let x be a new node;
x.key = k;
x.next = D.nil.next;
D.nil.next = x;
```

Algorithm 9: DELETE(D, k)

```
prev = L.nil;
while $prev.next.key \neq k$ do
    if $prev.next == L.nil$ then
        error "Key does not exist"
    end
    prev = prev.next;
end
prev.next = x.next;
```

Algorithm 10: SEARCH(D, k)

```
x = L.nil.next;
while $x \neq L.nil$ and $x.key \neq k$ do
    x = x.next;
end
return x;
```

INSERT $O(1)$, DELETE $O(n)$, SEARCH $O(n)$.

10.2-6.
Just link the tail of the first list to the head of the second list.

10.2-7.

Algorithm 11: LIST-REVERSE(L)

```
a = NIL;
b = L.head;
while $b \neq NIL$ do
    c = b.next;
    b.next = a;
    a = b;
    b = c;
end
L.head = a;
```

10.2-8.
To traverse the list, we initially have to know the addresses of two consecutive nodes, namely A and B. (If the element x have $x.np = 0$, we can conclude that there is only one element, so these cases can be figured out) If we have $B = A.next$, then $B.next = A \oplus B.np$. Forward direction traversal can be done in this manner. To traverse in backward direction, we can get $A.prev = B \oplus A.np$. In this manner, we can detect the head and the tail of the list. Using this fact, SEARCH, INSERT and DELETE can be implemented as follows:

Algorithm 12: LIST-SEARCH(L, k)

```
prev = 0;
x = L.head;
while $x \neq 0$ and $x.key \neq k$ do
    next = prev ⊕ x.np;
    prev = x;
    x = next;
end
return x;
```

Algorithm 13: LIST-INSERT(L, k)

```
Let x be a new node;
x.np = L.tail;
x.key = k;
L.tail.np = L.tail.np ⊕ x;
L.tail = x;
```

Algorithm 14: LIST-DELETE(L, k)

```
y = L.head;
prev = 0;
while $y \neq 0$ do
    next = prev $\oplus$ y.np;
    if $y \neq x$ then
        prev = y;
        y = next;
    else
        if $prev \neq 0$ then
            prev.np = prev.np $\oplus$ y $\oplus$ next;
        else
            L.head = next;
        end
        if $next \neq 0$ then
            next.np = next.np $\oplus$ y $\oplus$ prev;
        else
            L.tail = prev;
        end
    end
end
```

Reversing the list can be done by just swapping the order of initial two consecutive nodes.

10.3-1.
Multiple-array representation: next $\{2, 3, 4, 5, 6, null\}$, key $\{13, 4, 8, 19, 5, 11\}$, prev $\{null, 2, 3, 4, 5, 6\}$.
Single-array representation: $\{13, 4, null, 4, 7, 1, 8, 10, 4, 19, 13, 7, 5, 16, 10, 11, null, 13\}$.

10.3-2.
Let the object have size N.

Algorithm 15: ALLOCATE-OBJECT()

```
if free == NIL then
    error "out of space";
else
    x = free;
    free = x + N;
    return x;
end
```

Algorithm 16: FREE-OBJECT(x)

```
x + N = free;
free = x;
```

10.3-3.
Because free list acts like a stack, so there is no need to control the prev attribute.

10.3-4.
Let the free list be F, implemented as an array implementation of a stack. Then ALLOCATE-OBJECT and FREE-OBJECT will be as follows:

Algorithm 17: ALLOCATE-OBJECT()

```
if STACK-EMPTY(F) then
    error "out of space";
else
    x = POP(F);
    return x;
end
```

Algorithm 18: FREE-OBJECT(x)

```
p = F.top - 1;
p.prev.next = x;
p.next.prev = x;
x.key = p.key;
x.prev = p.prev;
x.next = p.next;
PUSH(F, p);
```

10.3-5.

Algorithm 19: COMPACTIFY-LIST(L, F)

```
index = 1;
curr = A[L.head];
while $curr.next \neq NIL$ do
    dest = A[index];
    SWAP(curr.prev.next, dest.prev.next);
    SWAP(curr.prev, dest.prev);
    SWAP(curr.next.prev, dest.next.prev);
    SWAP(curr.next, dest.next);
    if $F.head == index$ then
        F.head = curr;
    end
    curr = A[curr].next;
    index += 1;
end
```

10.4-1.
Root(18) has children (left: 12, right: 10), whose children are (left:7, right:4), (left:2, right:21) respectively. Node 4 has a left child 5.

10.4-2.

Algorithm 20: BINARY-TREE-INORDER(T)

```
x = T.root;
if $x \neq NIL$ then
    BINARY-TREE-INORDER(x.left);
    PRINT(x.key);
    BINARY-TREE-INORDER(x.right);
end
```

10.4-3.

Algorithm 21: BINARY-TREE-INORDER-ITERATIVE(T, S)

```
PUSH(S, T.root);
while !STACK-EMPTY(S) do
    x = PEEK(S);
    while $x \neq$ NIL do
        PUSH(S, x.left);
        x = PEEK(S);
    end
    POP(S);
    if !STACK-EMPTY(S) then
        x = POP(S);
        PRINT(x.key);
        PUSH(S, x.left);
    end
end
```

10.4-4.

Algorithm 22: NARY-TREE-TRAVERSAL(T)

```
x = T.root;
if $x \neq$ NIL then
    PRINT(x.key);
    curr = x.left-child;
    while curr $\neq$ NIL do
        NARY-TREE-TRAVERSAL(curr);
        curr = curr.right-sibling;
    end
end
```

10.4-5.

Algorithm 23: BINARY-TREE-ITERATIVE-INPLACE(T)

```
pred = FROM˙PARENT;
curr = T.root;
while curr ≠ NIL do
    if pred == FROM˙PARENT then
        PRINT(curr.key);
        if curr.left ≠ NIL then
            curr = curr.left;
            pred = FROM˙PARENT;
        else
            pred = FROM˙LEFT˙CHILD;
        end
    else if pred == FROM˙LEFT˙CHILD then
        if curr.right ≠ NIL then
            curr = curr.right;
            pred = FROM˙PARENT;
        else
            pred = FROM˙RIGHT˙CHILD;
        end
    else
        if curr.parent == NIL then
            curr = NIL;
            return;
        end
        if curr == curr.parent.left then
            pred = FROM˙LEFT˙CHILD;
        else
            pred = FROM˙RIGHT˙CHILD;
        end
        curr = curr.parent;
    end
end
```

10.4-6.
Use the boolean flag to identify the rightmost sibling. Let the right-sibling of the rightmost sibling points to its parent.

10-1.

Unsorted, singly linked list:
SEARCH $\Theta(n)$, INSERT $\Theta(1)$, DELETE $\Theta(n)$, SUCCESSOR $\Theta(n)$, PREDECESSOR $\Theta(n)$, MINIMUM $\Theta(n)$, MAXIMUM $\Theta(n)$.
Sorted, singly linked list:
SEARCH $\Theta(n)$, INSERT $\Theta(n)$, DELETE $\Theta(n)$, SUCCESSOR $\Theta(1)$, PREDECESSOR $\Theta(n)$, MINIMUM $\Theta(1)$, MAXIMUM $\Theta(n)$.
Unsorted, doubly linked list:
SEARCH $\Theta(n)$, INSERT $\Theta(1)$, DELETE $\Theta(1)$, SUCCESSOR $\Theta(n)$, PREDECESSOR $\Theta(n)$, MINIMUM $\Theta(n)$, MAXIMUM $\Theta(n)$.
Sorted, doubly linked list:
SEARCH $\Theta(n)$, INSERT $\Theta(n)$, DELETE $\Theta(1)$, SUCCESSOR $\Theta(1)$, PREDECESSOR $\Theta(1)$, MINIMUM $\Theta(1)$, MAXIMUM $\Theta(1)$.

10-2.
a.
MAKE-HEAP : Just create a new, empty linked list.
INSERT : Perform a linear scan to search where to insert an element so that the list remains sorted. This takes linear time.
MINIMUM : First element is minimum, so it takes constant time.
EXTRACT-MIN : First element is minimum, so it takes constant time.
UNION : Equivalent to merge operation between two sorted lists. This takes linear time with respect to the sum of the lengths of the two lists.
b.
MAKE-HEAP : Just create a new, empty linked list.
INSERT : To insert an element x to the heap, perform a linear scan from the head of the list until the first node y which is strictly larger than x is found. If no such element exists, insert x at the tail. If such y exists, replace y by x, and insert y, by performing a linear scan from the node following x. Do this recursively. Since we check each node at most once, this takes linear time.
MINIMUM : First element is minimum, so it takes constant time.
EXTRACT-MIN : First element is minimum, and we need to swap it with its child, the smaller one. The swapped child must be replaced with its child, the smaller one as well. Repeat this until we reach the bottom of the heap. Since we traverse each node at most once, this takes linear time.
UNION : Can be implemented as follows:

Algorithm 24: UNION(A, B)

```
if $A.head == NIL$ then
    return B;
end
a = A.head;
while $B.head \neq NIL$ do
    if $B.head.key \leq a.key$ then
        INSERT(B, a.key);
        a.key = B.head.key;
        EXTRACT-MIN(B);
    end
    a = a.next;
end
A.tail.next = B.head;
B.head.prev = A.tail;
return A;
```

This takes linear time in the lengths of the sum of the two lists.
c.
The algorithms in part (b) don't require the elements to be distinct, so we can use the same ones.

10-3.
a. If COMPACT-LIST-SEARCH takes t iterations, then we have that it takes at most t skips to get the desired value, so COMPACT-LIST-SEARCH' yields the same answer. Since each iteration in COMPACT-LIST-SEARCH correpsonds to a possible random jump followed by a step through traversing the linked list, the total number of iterations of both the for and while loops in COMPACT-LIST-SEARCH' is at least t.
b. The for loop runs exactly t times, and the while loop runs exactly X_t times. Hence the total expected running time is $O(t + \mathbb{E}[X_t])$.
c. Since the equation C.25, we have $\mathbb{E}[X_t] = \sum_{i=1}^{\infty} P(X_t \geq i)$. So, it suffices to show that $P(X_t \geq i) \leq (1 - \frac{i}{n})^t$. Since $X_t \geq i$ means that every random choice among t-times selection results picking an element that is either at least i steps before the desired key, or is after the desired key. Since there are n - i such elements, and each selection is independent, the overall probability is at most $\frac{n-i}{n}^t$.

d.

$$\sum_{r=0}^{n-1} r^t \leq \int_0^n r^t dr = \frac{n^{t+1}}{t+1}.$$

e.

$$\mathbb{E}[X_t] = \sum_{r=1}^{n}(1-\frac{r}{n})^t = \sum_{r=1}^{n}\sum_{i=0}^{t}\binom{t}{i}(-\frac{r}{n})^i = \sum_{i=0}^{t}\sum_{r=1}^{n}\binom{t}{i}(-\frac{r}{n})^i$$

$$= \sum_{i=0}^{t}\binom{t}{i}(-1)^i\frac{1}{n}(n^i - 1 + \sum_{r=0}^{n-1} r^t) \leq \sum_{i=0}^{t}\binom{t}{i}(-1)^i\frac{1}{n}(n^i - 1 + \frac{n^{i+1}}{i+1})$$

$$= \frac{(1-n)^t}{n} + \sum_{i=0}^{t}\binom{t}{i}(-1)^i(-\frac{1}{n} + \frac{n^i}{i+1}) \leq \sum_{i=0}^{t}\binom{t}{i}(-1)^i\frac{n^i}{i+1}$$

$$= \frac{1}{t+1}\sum_{i=0}^{t}\binom{t+1}{i+1}(-n)^i \leq \frac{(1+n)^{t+1}}{t+1}.$$

f. Combining parts b. and e. gives the expected running time $O(t + \frac{n}{t+1}) = O(t + \frac{n}{t})$.
g. Since COMPACT-LIST-SEARCH minimizes the running time of COMPACT-LIST-SEARCH', the minimized expected running time happens at $t = \sqrt{n}$, giving the expected running time $O(\sqrt{n})$.
h. The algorithm is able to skip elements only if it encountered an element bigger than our current key.