III Data Structures

10: Elementary Data Structures

10.1: Stacks & queue

STACK — LIFO → PUSH, POP

QUEUE — FIFO → ENQUEUE, DEQUEUE

```
STACK-EMPTY(S)
① if S.top == 0
②     return TRUE
③ else return FALSE

PUSH(S, x)
① S.top = S.top + 1
② S[S.top] = x

POP(S)
① if STACK-EMPTY(S)
②     error "underflow"
③ else S.top = S.top - 1
④     return S[S.top + 1]
```

$T(n) = O(1)$

Queue is empty : Q.head = Q.tail

Initially Q.head = Q.tail = 1

Queue full : Q.head = Q.tail + 1

n = Q.length

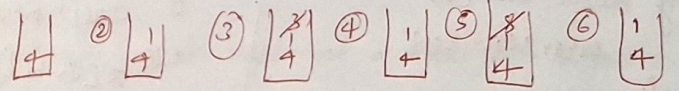
```
ENQUEUE(Q, x)
① Q[Q.tail] = x
② if Q.tail == Q.length
③     Q.tail = 1
④ else Q.tail = Q.tail + 1

DEQUEUE(Q)
① x = Q[Q.head]
② if Q.head == Q.length
③     Q.head = 1
④ else Q.head = Q.head + 1
⑤ return x
```

$T(n) = O(1)$

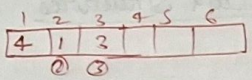
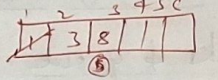
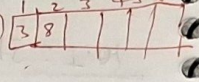
Exercise

10.1-1:



10.1-2 : The first stack starts at 1 & grows up towards n, while the second starts from n & grows down towards 1.

Stack overflow happens when elements is pushed when the 2 stack pointers are adjacent.

10.1-3: ①  ④  ⑥ 

10.1-4: ENQUEUE & DEQUEUE to detect underflow & overflow of a queue
QUEUE-EMPTY, QUEUE-FULL

```
* QUEUE-EMPTY(Q)
    if Q.head == Q.tail
        return TRUE
    else return FALSE

* QUEUE-FULL(Q)
    if Q.head == Q.tail+1 or (Q.head == 1
                              && Q.tail == Q.length)
        return TRUE
    else return FALSE

*ENQUEUE(Q,x)
    if QUEUE-FULL(Q)
        error "overflow"
    else
        Q[Q.tail] = x
        if Q.tail == Q.length
            Q.tail = 1
        else Q.tail = Q.tail+1

* DEQUEUE(Q)
    if QUEUE-EMPTY(Q)
        error "underflow"
    else
        x = Q[Q.head]
        if Q.head == Q.length
            Q.head = 1
        else Q.head = Q.head+1
        return x
```

10.1-5: deque (double-ended queue) implementation

```
*HEAD-ENQUEUE(Q,x)
    if QUEUE-FULL(Q)
        error "overflow"
    else
        if Q.head == 1
            Q.head = Q.length
        else Q.head = Q.head-1
        Q[Q.head] = x

* TAIL-ENQUEUE(Q,x)
    if QUEUE-FULL(Q)
        error "overflow"
    else
        Q[Q.tail] = x
        if Q.tail == Q.Length
            Q.tail = 1
        else Q.tail = Q.tail+1

* HEAD-DEQUEUE(Q)
    if QUEUE-EMPTY(Q)
        error "underflow"
    else
        x = Q[Q.head]
        if Q.head == Q.length
            Q.head = 1
        else Q.head = Q.head+1
        return x

* TAIL-DEQUEUE(Q)
    if QUEUE-EMPTY(Q)
        error "underflow"
    else
        if Q.tail == 1
            Q.tail = Q.length
        else Q.tail = Q.tail-1
        x = Q[Q.tail]
        return x
```

10.1-6: Show how to implement a queue using 2 stacks. Analyze the running time of the queue operations.

→ ENQUEUE : $\Theta(1)$
→ DEQUEUE: Worst $\Theta(n)$, amortized $\Theta(1)$.

Let the 2 stacks A & B.

ENQUEUE pushes elements on B. DEQUEUE pops elements from A. If A is empty, the contents of B are transfered to A by popping them out of B & pushing them to A. That way they appear in reverse order & are popped in the original.

A DEQUEUE operation can perform in $\Theta(n)$, but that will happen only when A is empty. If many ENQUEUEs & DEQUEUEs are performed, the total time will be linear to the number of elements, not to the largest length of the queue.

10.1-7: Show how to implement a stack using 2 queue. Analyze the running time of the stack operations.

→ PUSH : $\Theta(1)$
→ POP : $\Theta(n)$

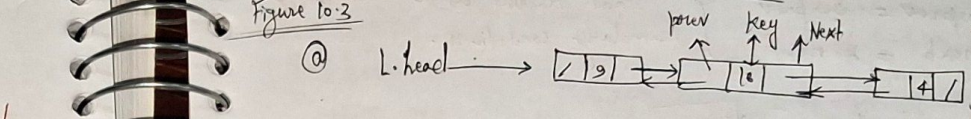
We have 2 queue & mark one of them as active. PUSH queues an element on

on the active queue. POP should dequeue all but one element of the active queue & queue them on the inactive. The roles of the queues are then reversed & the final element left in the (now) inactive queue is returned.

The PUSH operation is $\Theta(1)$, but the POP operation is $\Theta(n)$ where n is the number of elements in the stack.

10.2 : Linked Lists

Figure 10.3



* $x.next$ → successor
* $x.prev$ → predecessor
* First element on head
* If $x.next = NIL$ ⇒ No successor or Last element or tail
* If $x.prev \Rightarrow NIL$ } No predecessor or First element or Head.
* If $L.head = NIL$ } list is empty.

Searching a linked list

LIST-SEARCH (L,k)

① $x = L.head$
② while $x \neq NIL$ && $x.key \neq k$
③ $x = x.next$
④ return x

$\Theta(n)$ times in worst case, since it may have to search the entire list

Insert into a linked list

LIST-INSERT (L,x)

① $x.next = L.head$
② if $L.head \neq NIL$
③ $L.head.prev = x$
④ $L.head = x$
⑤ $x.prev = NIL$

} $O(1)$

Deleting from a linked list

LIST-DELETE (L,x)
if $x.prev \neq NIL$
$x.prev.next = x.next$
else $x.next \neq NIL$
$x.next.prev = x.prev$

Sentinels The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head & tail of the list :

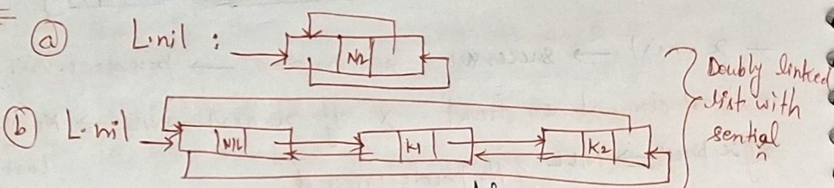
FILE 22 HASHING INTRO 1 DATE

LIST-DELETE'(L,x)

① x.next.prev = x.prev
② x.prev.next = x.next

} A sentinel is a dummy object that allows us to simplify boundary conditions.

Figure 10.4
Ex:-



then LIST-SEARCH'(L,k)

① x = L.nil.next
② while x ≠ L.nil && x.key ≠ k
③ x = x.next
④ return x

&& LIST-INSERT'(L,x)

① x.next = L.nil.next
② L.nil.next.prev = x
③ L.nil.next = x
④ x.prev = L.nil

Exercise:

10.2-1: Can you implement the dynamic-set operation INSERT on a singly linked list in O(1) time? How about DELETE?

* LIST-INSERT(L,x)
  x.next = L.head
  L.head = x

DELETE: you can copy the value from successor to element you want to delete & then you delete the successor in O(1) time. This sol^n is not good in situations when have a large object, in that case copying the whole object will be a bad idea.

10.2-2: Stack using singly linked list L:

STACK-EMPTY(L)
if L.head == NIL
return TRUE
else return FALSE

PUSH(L,x)
x.next = L.head
L.head = x

POP(L)
if STACK-EMPTY(L)
error "underflow"
else
x = L.head
L.head = L.head.next
return x

18 Nov

10.2-3 Queue by singly linked list L.

QUEUE-EMPTY(L)
if L.head == NIL
return TRUE
else return FALSE

ENQUEUE(L,x)
if QUEUE-EMPTY(L)
L.head = x
else L.tail.next = x
L.tail = x
x.next = NIL

DEQUEUE(L)
if QUEUE-EMPTY(L)
error "underflow"
else
x = L.head
if L.head == L.tail
L.tail = NIL
L.head = L.head.next
return x

10.2-4: LIST-SEARCH'(L,K)
x = L.nil.next
L.nil.key = K
while x.key != K
x = x.next
return x

10.2-5:

INSERT: O(1)
LIST-INSERT''(L,x)
x.next = L.nil.next
L.nil.next = x

DELETE: O(n)
LIST-DELETE''(L,x)
prev = L.nil
while prev.next != x
if prev.next == L.nil
error "element not exist"
prev = prev.next
prev.next = x.next

SEARCH: O(n)
LIST-SEARCH''(L,k)
x = L.nil.next
while x != L.nil
&& x.key != k
x = x.next
return x

10.2-6: If both sets are a doubly linked lists, we just point link the last element of the first list to the first element in the second. If the implementation uses sentinels, we need to destroy one of them.

LIST-UNION(L[1], L[2])
L[2].nil.next.prev = L[1].nil.prev
L[1].nil.prev.next = L[2].nil.next
L[2].nil.prev.next = L[1].nil
L[1].nil.prev = L[2].nil.prev

10.2-7:

```
LIST-REVERSE(L)
  p[1] = NIL
  p[2] = L.head
  while p[2] != NIL
    p[3] = p[2].next
    p[2].next = p[1]
    p[1] = p[2]
    p[2] = p[3]
  L.head = p[1]
```

$\Theta(n)$ times & constant space.

10.2-8:

```
*LIST-SEARCH(L,k)
  prev = NIL
  x = L.head
  while x != NIL and x.key != k
    next = prev XOR x.np
    prev = x
    x = next
  return x
```

```
*LIST-INSERT(L,x)
  x.np = NIL XOR L.tail
  if L.tail != NIL
    L.tail.np = (L.tail.np XOR NIL) XOR x
  if L.head == NIL
    L.head = x
  L.tail = x
```

```
*LIST-DELETE(L,x)
  y = L.head
  prev = NIL
  while y != NIL
    next = prev XOR y.np
    if y != x
      prev = y
      y = next
    else
      if prev != NIL
        prev.np = (prev.np XOR y) XOR next
      else L.head = next
      if next != NIL
        next.np = prev XOR (y XOR next.np)
      else L.tail = prev
```

```
*LIST-REVERSE(L)
  temp = L.head
  L.head = L.tail
  L.tail = temp.
```

10.3: Implementing pointers & objects

* Garbage collector is resposible for determining which objects are unused.
* We keep the free objects in a singly linked list, which call the free list.

```
ALLOCATE-OBJECT()
  if free == NIL
    error "out of space"
  else x = free
    free = x.next
  return x
```

```
FREE-OBJECT(x)
  x.next = free
  free = x.
```

Excercise:

10.3-1: Given sequence $\langle 13, 4, 8, 19, 5, 11 \rangle$

* A multiple-array representation with $L = 2$,

Index	1	2	3	4	5	6	7
next	/	3	4	5	6	7	/
key	/	13	4	8	19	5	11
prev	/	/	2	3	4	5	6

* A Single-array version with $L = 1$

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
key	13	4	/	4	7	1	8	10	4	19	13	7	5	16	10	11	/	13

10.3-2:

```
ALLOCATE-OBJECT()
  if free == NIL
    error "Out of space"
  else x = free
    free = A[x+1]; return x
```

```
FREE-OBJECT(x)
  A[x+1] = free
  free = x
```

Ans:

10.3-3: We implement ALLOCATE-OBJECT & FREE-OBJECT in the hope of managing the storage of currently non-used object in the free list so that one can be allocated for reusing. As the free list acts like a stack, to maintain this stack-like collection, we merely remember its first pointer & set the next attribute of objects. There is no need to worry the prev attribute, for it hardly has any impact on the resulting free list. (Use array implementation of stack)

10.3-4:

```
ALLOCATE-OBJECT()
  if STACK-EMPTY(F)
      error "out of space"
  else x = POP(F)
  return x
```

```
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:

We combine the array key, prev, & next by a multiple-array A. Each object of A's is either in List L or in the free list F, but not in the both. The procedure COMPACTIFY-LIST transpose the

```
COMPACTIFY-LIST(L, F)
  TRANSPOSE(A[L.head], A[1])
  if F.head == 1
      F.head = L.head
  L.head = 1
  l = A[L.head].next
  i = 2
  while l != NIL
      TRANSPOSE(A[l], A[i])
      if F == i
          F = l
      l = A[l].next
      i = i+1
```

```
TRANSPOSE(a, b)
  Swap(a.prev.next,
       b.prev.next)
  Swap(a.prev, b.prev)
  Swap(a.next.prev,
       b.next.prev)
  Swap(a.next, b.next)
```

First object in L with the object in A, the second objects until the list L is exhausted.

10.4: Representing rooted trees

*** Binary trees**

→ Attributes p, left, & right to store pointer to the parent, left child & right child of each node in a binary tree T.

→ If x.p = NIL } x is root.

→ If x.left = NIL } No left child, similary for the right child.

→ If T.root = NIL } the tree is empty.

*** Rooted trees with unbounded branching**

→ There is an advantage of using only O(n) space for any n-node rooted tree.

→ Each node x has only 2 pointers:
- x.left-child } points to the leftmost child of node x &
- x.right-sibling } points to the sibling of x immediately to its right.

→ x.left-child = NIL } No children.

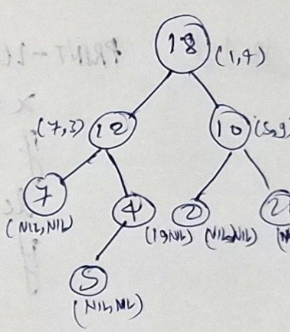
→ x.right-sibling = NIL } node x is rightmost child of its parent.

Exercise

10.4-1: Given

index	key	left	right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

Draw rooted at index 6



10.4-2:

```
PRINT-BINARY-TREE(T)
    x = T.root
    if x != NIL
        PRINT-BINARY-TREE(x.left)
        print x.key
        PRINT-BINARY-TREE(x.right).
```

10.4-3:

```
PRINT-BINARY-TREE(T,S)
    PUSH(S, T.root)
    while !STACK-EMPTY(S)
        x = S[S.top]
        while x != NIL
            PUSH(S, x.left)
            x = S[S.top]
        POP(S)
        if !STACK-EMPTY(S)
            x = POP(S)
            print x.key
            PUSH(S, x.right)
```

10.4.4:

```
PRINT-LCRS-TREE(T)
    x = T.root
    if x != NIL
        lc = x.left-child
        if lc != NIL
            PRINT-LCRS-TREE(lc)
            rs = lc.right-sibling
            while rs != NIL
                PRINT-LCRS-TREE(rs)
                rs = rs.right-sibling
```

* 10.4-5:

```
PRINT-KEY(T)
    prev = NIL
    x = T.root
    while x != NIL
        if prev = x.parent
            print x.key
            if x == x.left
                x.left
            else
                if x.right
                    x.right
                else x.parent
        else if prev == x.left && x.left != NIL
            prev = x
            x = x.right
        else
            prev = x
            x = x.parent.
```

* 10.4-6: Use boolean to identify the last sibling of the last sibling's right-sibling points to the parent.