School of ECE, UNIST
CSE221 (Data Structures)
Spring 2019

Instructor:
Hyungon Moon

Midterm, 21:00 - 23:00 (120min) Apr 22, 2019

Your Name (In English):	
Your Student ID:	

Instruction:

1. You will be submitting all the sheets that you are given.
2. You can neither ask questions nor leave the room within the first 45 minutes of the exam.
3. Write your student ID at every page, bottom left.
4. Your answers should be printed at the designated locations, and written with an **inerasable black or blue** pen.
5. You are not allowed to go to the rest room and come back.

Question	Points	Score
1	4	
2	10	
3	5	
4	18	
5	3	
6	12	
7	14	
8	18	
9	23	
10	23	
11	10	
Total:	140	

Enjoy!

1. **Handling polynomials with arrys**

   A polynomial in a single indeterminate x, degree d can be represented like this:

   $$P(x) = \sum_{k=0}^{d} a_k x^k$$

   Such a polynomial can have $d+1$ terms at most, but the number of terms n coul be smaller than $d+1$

   In this question, we will represent such polynomials in two ways, depending on the expected behavior of our program. Assume that the polynomials in the two programs are static; none of them modifies an already created polynomial. This allows us two represent the polynomials using arrays.

   (a) [2 points] The first program is handling short (small number of terms n) but high order polynomials (large degree d). For this reason, it is desired to use a representation whose space complexity is $O(n)$, if possible. Fill out the blank of the following `struct` definition assuming that we are using a dynamically allocated array of the `struct` to represent the polynomial. Assume that we have a type called `BigInt` which can represent an arbitrary integer.

   ```
   struct element_1 {
   ```

   Solution:

   ```
       BigInt degree;
       BigInt coefficient;
   ```

   ```
   };
   ```

   (b) [2 points] The second program is handling long (large number of terms n) which is similar tot the order polynomials d. For this reason, it is desired to use a representation whose space complexity is $O(d)$. Fill out the blank of the following `struct` definition assuming that we are using a dynamically allocated array of the `struct` to represent the polynomial. Assume that we have a type called `BigInt` which can represent an arbitrary integer.

   ```
   struct element_1 {
   ```

   Solution:

   ```
       BigInt coefficient;
   ```

   ```
   };
   ```

2. **Linked List Implementation.**

Take a look at this code snippet to answer the questions.

```
1  struct Node {
2      int value;
3      Node* next;
4      Node* prev;
5  };
6  class List {
7      Node* head;
8      Node* tail;
9      bool insert(Node* n, int val);
10     bool remove(Node* n);
11 };
```

(a) [5 points] Implement the `remove` function that is declared above. `remove` returns false if it fails to find the `Node` to be removed.

Solution:

```
bool remove(Node* n) {
  Node* p = head;
  while(p != n) {
    p = p->next;
  }
  if(p == nullptr) return false;
  if(p->prev) p->prev->next = p->next;
  if(p->next) p->next->prev = p->prev;
  delete p;
  return true;
}
```

Criteria

- Traversing: 1 point
- Handling prev correctly: 1 point
- Handling next correctly: 1 point
- Deleting the memory: 1 point
- Correct implementation: 1 point

Question 2 continues on the next page...

(b) [5 points] Implement the `insert` function that is declared above. It inserts new `Node` after the given `Node`. `remove` returns false if it fails to find the given `Node`.

Solution:

```
bool insert(Node* n, int val) {
  Node* p = head;
  while(p != n) {
    p = p->next;
  }
  if(p == nullptr) return false;
  Node* q = new Node();
  q->value = val; q->prev = p; q->next = p->next;
  p->next = q;
  if(p->next) p->next->prev = q;
  return true;
}
```

Criteria

- Traversing: 1 point
- Handling prev correctly: 1 point
- Handling next correctly: 1 point
- Initializing the memory: 1 point
- Correct implementation: 1 point

3. [5 points] **Stack**

   Thinks of an array-based stack. Though the cost of `pop` will have the time complexity of $O(1)$, `push` is not, if we cannot tell the maximal length of the stack in advance. When we use the doubling strategy, in which we double the size of an internal array whenever it's full, state and argue about the amortised time complexity of `push`.

Solution: We calculate the amortised number of primitive operations while we perform n `push`s. Each `push` costs 1 data copy, and we should have doubled the array for $C = \lfloor \log_2 \frac{n}{c} \rfloor$ times, where c is the initial size of the stack. As each doubling comes with the cost of copying all the original elements, the number of total copies is $\sum_{i=0} Cc2^i =$. Summing these up, we get

$$T(n) = n + 2c(2^{\lfloor \log_2 \frac{n}{c} \rfloor} - 1) <= n + 2n = 3n \text{ (for inserting n items).}$$

$$T(n) = O(n)$$

$$\text{For one push, } T(n) = O(1)$$

Criteria

1. State $O(1)$: 1 point.
2. Correct informal analysis: 3 points.
3. Correct formal analysis: 1 point.

4. **Circular Queue**

Answer the questions regarding this incomplete implementation of a circular queue.

```
#define N 1000
class cq {
    int array[N];
    size_t front, back;
    cq() {front = 0; back = 0;}
    bool isEmpty() { return tail == head; }
    bool isFull();
    bool push(int val);
    bool pop();
    int top() {
        if (isEmpty()) raise std::runtime_error("empty");
        return array[front];
    }
};
```

(a) [4 points] Implement the `isFull` function. It returns true if the queue is empty and returns false otherwise.

Solution:

```
bool isEmpty() {
  return (front + 1) % N == back;\
}
```

Criteria

1. `front + 1 == back`: 2 points
2. `(front + 1) % N == back`: 4 points
3. No modular operation: -1 point

(b) [6 points] Implement the `push` function. It returns true if it could push the value, and returns false otherwise.

Solution:

```
bool push(int val) {
  if (isFull()) return false;
  array[back] = val;
  back += 1;
  back %= N;
  return true;
}
```

Criteria

1. Checking if full: 1 point
2. Writing to back: 2 points

Question 4 continues on the next page...

3. Modular operation: 2 points
4. Returning true: 1 point

Question 4 continues on the next page...

(c) [3 points] Calculate the time complexity of `push` function.

Solution: `O(1)`

(d) [5 points] Consider that we have a circular queue with length 5 as follows.

Front	2
Back	4

2	3	5	7	13

Illustrate the resulting circular queue after the following sequence of operations.

```
push(17);
pop();
pop();
pop();
push(19);
push(23);
```

Solution:

Front	0
Back	2

19	23	5	7	17

Criteria

- Queue elements: 0.5 each
- front/back: 1 each

5. [3 points] **StackQueue**

This question is about implementing a queue using a stack. Implement the `pop` function using the given interfaces to the stack. Do nothing if the queue is empty.

```
1  class stack {
2      virtual void push(int val);
3      virtual void pop();
4      virtual int top();
5      virtual int isEmpty();
6  };
7
8  class stackqueue {
9      class stack sa, sb;
10     void push(int val) {
11         sa.push(val);
12     }
13     void pop();
14 }
```

Solution:

```
void pop() {
  if(sb.isEmpty()) {
    if(sa.isEmpty()) return;
    while(!sa.isEmpty()) {
      sb.push(sa.top())
      sa.pop();
    }
  }
  sb.pop();
}
```

6. **Performance Analysis**

(a) [3 points] State the formal definition of the big-O notation, $O(x)$.

> **Solution:** Given $f(x)$ and $g(x)$, we say that $f(x) = O(g(x))$ if there exists a positive constant c and x_0 such that $f(x) <= cg(x)$ for all $x >= x_0$.

(b) [3 points] State the formal definition of the theta notation, $\Theta(x)$.

> **Solution:** Given $f(x)$, $g(x)$, we say that $f(x) = \Theta(g(x))$ if there exists a positive constant c_1, c_2 and x_0 such that $c_1 g(x) <= f(x) <= c_2 g(x)$ for all $x >= x_0$.

(c) [3 points] Calculate the worst case time complexity of this function. You should also state how you got the answer.

```
#define N 10000
int buffer[N];
int foo(int n) {
    int ret = 0;
    for(int i = 0; i < n; i += 1) {
        for(int j =i; j < i + 10; j += 1) {
            for(int k = i; k < n; k += 1) {
                ret += buffer[i%N] + buffer[j%N] + buffer[k&N];
            }
        }
    }
    return ret;
}
```

> **Solution:**
>
> - The loop with `k`: $n - i$ times.
> - The loop with `j`: 10 times.
> - The loop with `i`: n times. $T(n) = n \times 10 \times (n + n - 1 + \cdots + 1) = 5n(n-1) = O(n^2)$
>
> **Crietria**
>
> 1. Correct answer: 1 point.
> 2. Correct argument: 2 points.

Question 6 continues on the next page...

(d) [3 points] Calculate the worst case time complexity of this function. You should also state how you got the answer.

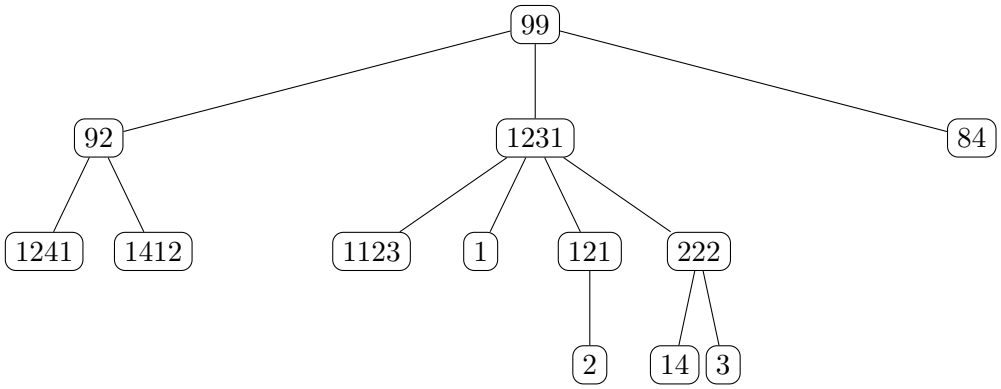
```
1 int bar(int n) {
2     if(n == 1 || n == 2) return 1;
3     return bar(n-1) + bar(n-2)
4 }
```

Solution:
The call graph can be represented as a tree with height n. Considering the maximal number of nodes in the tree, we get

Crietria

1. Correct answer: 1 point.
2. Correct argument: 2 points.

7. **Tree Basics**



(a) [8 points] Fill out the blanks about the given tree.

Degree	**4**
Height	**3**
The number of internal nodes	**6**
The set of ancestors of node 14	**{222, 1231, 99}**

(b) [2 points] State the result of pre-order traversal.

Solution: 99, 92, 1241, 1412, 1231, 1123, 1 121, 2, 222, 14, 3, 84

(c) [2 points] State the result of post-order traversal.

Solution: 1241, 1412, 92, 1123, 1, 2, 121, 14, 3, 222, 1231, 84, 99

(d) [2 points] State the result of level-order traversal.

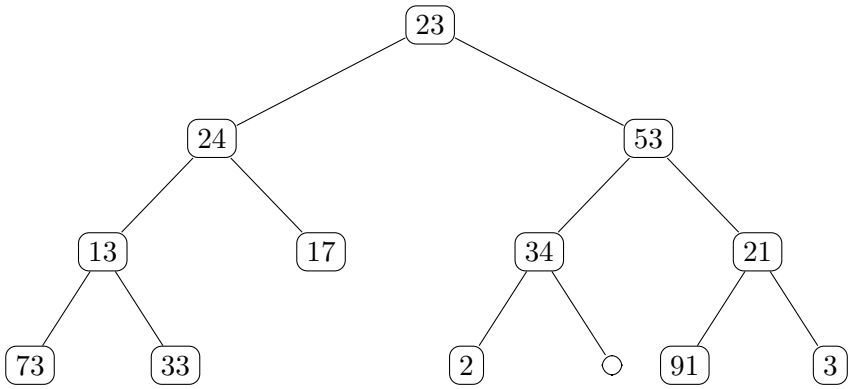
Solution: 99, 92, 1231, 84, 1241, 1412, 1123, 1, 121, 222, 2, 14, 3

8. **Binary trees**

(a) [5 points] Draw a binary tree from given pre- and in-order traversal.

- Pre-order: 23, 24, 13, 73, 33, 17, 53, 34, 2, 21, 91, 3
- In-order: 73, 13, 33, 24, 17, 23, 2, 34, 53, 91, 21, 3

Solution:



(b) [3 points] Implement the `left` functions of this class definition which uses an array to contain a binary tree with the standard indexing scheme. It returns `N` on failure.

```
1 template<size_t N>
2 class btree {
3     int array[N];
4     size_t left(size_t index i);
5 };
```

Solution:

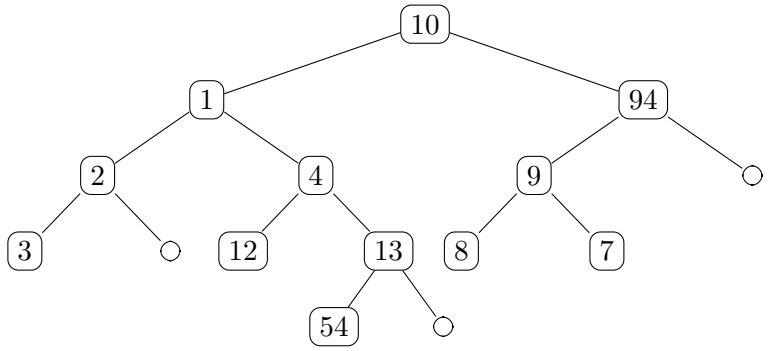
```
size_t left(size_t index i) {
  if(2 * i < N) return 2 * i;
  else return N;
}
```

Criteria: This version assuming the root at both 0 and 1 gets the full score.

```
size_t left(size_t index i) {
  if(2 * (i+1) < N) return (2 * i)-1;
  else return N;
}
```


Question 8 continues on the next page...

(c) [5 points] What is the smallest value of `N` to represent the following tree using the above array-based representation? State the value and illustrate the array containing the tree. An empty circle means there is no node.



Solution: N: 22

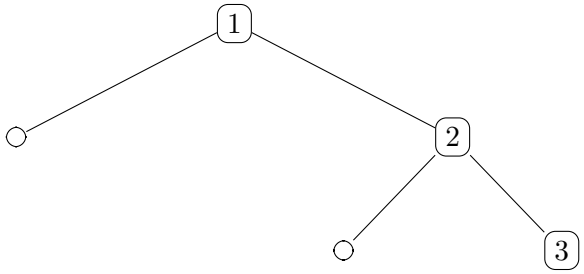
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
10	1	94	2	4	9		3		12	13	8	7									54	

Criteria

- N: 3 points
- Representation: 4 points
- An alternative version putting the root at index 1 also gets the full score.

(d) [5 points] What is the worst case asymptotic space complexity of an array-based representation when the size of a binary tree is n? Illustrate a worst case example when $n = 3$ (A tree and its representation).

Solution: $O(2^n)$



1		2				3

Criteria

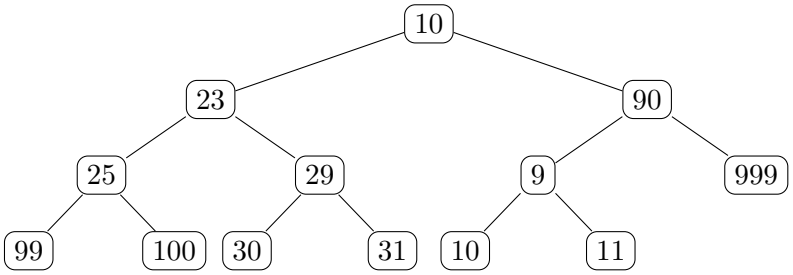
- Complexity: 2 points.
- Tree example (the same shape): 2 points.

Question 8 continues on the next page...

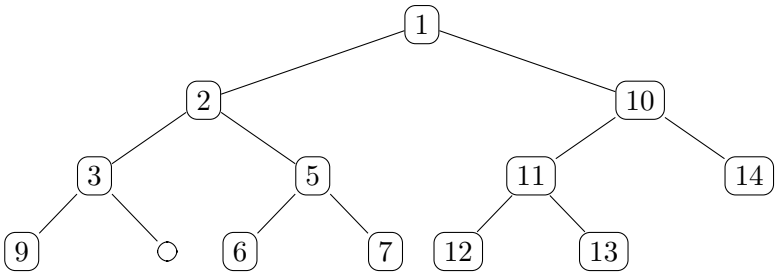
- Representation: 1 point.

9. **Heap**

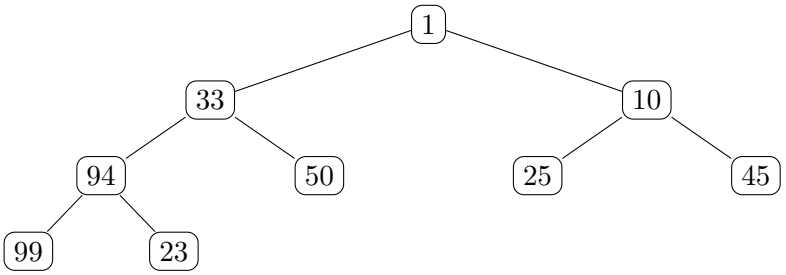
(a) [8 points] Determine if each of these tree is a heap or not.



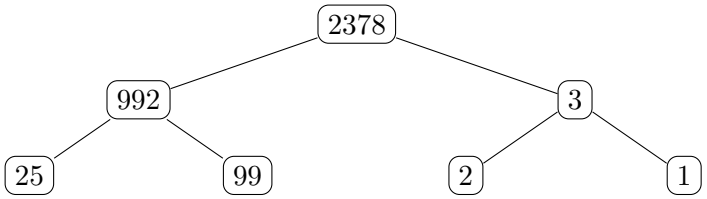
Answer (Yes/No): **No**



Answer (Yes/No): **Yes**



Answer (Yes/No): **No**



Answer (Yes/No): **Yes**

(b) [10 points] Implement the `push` function in the following code snippet implementing a max heap. Assume and maintain that `tail` points to the parent of the next empty node, if the heap is not empty.

```
1  class maxheap {
2    struct Node {
3      int val;
4      Node* left, right, parent;
5    };
6    Node* root;
7    Node* tail;
8    void push(int val);
9  };
```

Solution:

```
void push(int val) {
  Node* n = Node(); n->val = val;
  n->left = root->right = root->parent = nullptr;
  if(!root) {
    root = tail = n; return;
  }
  if(tail->left) tail->right = n;
  else tail->left = n;
  n->parent = tail;
  while(n == root || n->val <= n->parent->val) {
    int temp = n->parent->val;
    n->parent->val = n->val;
    n->val = temp;
    n = n->parent;
  }
  if(tail->right == n) {
    while(tail != root) {
      if(tail == tail->parent->left) {
        tail = tail->parent->right;
        break;
      }
      else tail = tail->parent;
    }
    while(tail->left != nullptr) tail = tail->left;
  } }
```

Criteria

- Handling an empty heap: 2 points
- Initializing left/right/parent of the new node correctly: 1 point each (total 3 points)
- Inserting to the last empty spot: 1 point
- Correctly bubbling up: 2 points
- Fixing tail: 2 points

Question 9 continues on the next page...

(c) [5 points] Calculate the asymptotic time complexity of the `push` in terms of the number of nodes n, with the argument.

Solution:

- The asymptotic time complexity of each `while` loop is $O(h)$ in which h is the height of the heap. (2 points)
- Heap is always a complete tree, so h = $O(\log n)$ (2 points)
- Answer: $O(\log n)$ (1 points)

10. **Map**

We are given a sequence of integers as follows.

1,5,3,22,2,34,24,7,53,28,85,35,36,49

(a) [7 points] Illustrate a hash table with seven buckets, each having three slots, when we insert the above integers in sequence. We use a simple hash function `h(x) = x % 7` and use linear probing to handle overflows.

Index	slot 1	slot 2	slot 3
0	**7**	**28**	**35**
1	**1**	**22**	**85**
2	**2**	**36**	**49**
3	**3**	**24**	
4	**53**		
5	**5**		
6	**34**		

Solution: 0.5 point each.

(b) [2 points] Calculate the loading density of the above hash table.

Solution: $14/21 = 0.67$

(c) [7 points] Illustrate a hash table with the same hash function but using chaining instead of linear probing.

Solution:

0 → 7 → 28 → 35 → 49

1 → 1 → 22 → 85 → 36

2 → 2

3 → 3 → 24

4 → 53

5 → 5

6 → 34

0.5 point each.

(d) [2 points] Calculate the loading density of the above hash table.

Solution: $14/7 = 2$

(e) [5 points] When the loading density of a hash table with chaining is α, compute the expected number of comparisons for successful search with the argument.

Solution:

The expected length of chain is α. When the item that we are looking for is at kth location, the number of comparisons required is k. Calculating the amortised, $\frac{1+2+\cdots+\alpha+\alpha}{\alpha} = \frac{(\alpha+2)\alpha}{2\alpha} = 1+\frac{\alpha}{2}$ so the expected number of comparisons for successful search is $\frac{\alpha}{2}+1$

Note: $\frac{1+\alpha}{2}$ is also a right answer.

Criteria

- Answer: 2 points
- Argument: 3 points

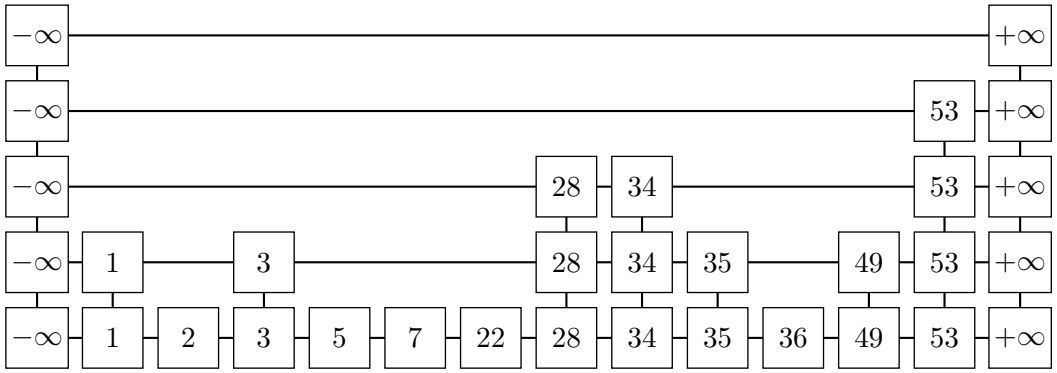
11. **Skip list**

We again consider this sequence of integers.

1,5,3,22,2,34,24,7,53,28,85,35,36,49

(a) [5 points] Illustrate the skip list generated from the sequence. Assume that after the random function output drives 1,3,35,49 to appear at two levels, 34, 28 at three levels, and 53 at four levels.

Solution:



1 Point per level.

(b) [3 points] State the sequence of nodes we are visiting when searching for 36. Represent 28 at level 3 as (28,3).

Solution: $(\infty, 5) \rightarrow (\infty, 4) \rightarrow (\infty, 3) \rightarrow (28, 3) \rightarrow (34, 3) \rightarrow (34, 2) \rightarrow (35, 2) \rightarrow (35, 1) \rightarrow (36, 1)$

(c) [2 points] What is the worst case asymptotic time complexity of search using a skip list with n elements? Provide an example also.

Solution: $O(n)$, An example should be a skewed one renders a search visiting every node.

- $O(n)$: 1 point.
- Example: 1 point.

The end of questions.