Exam 2 Review

Exam 2 Topics

- Stacks (coding & tracing)
- Queues (coding)
- Hashing (coding & tracing)
- Sorting (tracing & time complexity)
- Heaps (tracing)
- Recursion (coding)

Stacks #1

Write the function **`string postfixToInfix(string exp)`** which takes in a postfix expression and converts it to infix and returns the infix expression.

```
string postfixToInfix(string exp){




}
```

Stacks #1

```
string postfixToInfix(string exp){
    stack<string> s;
    string str1, str2, str3;
    for(int i = 0; i < exp.length(); i++){
        if(isdigit(exp[i]))
            s.push(exp.substr(i, 1));

        else{
            str1 = s.top(); s.pop();
            str2 = s.top(); s.pop();
            str3 = '(' + str2 + exp[i] + str1 + ')';
            s.push(str3);
        }
    }

    return s.top();
}
```

Stacks #2

Write the function **`stack<int> sortStack(stack<int> &st)`** which returns a sorted stack in descending order.

Note: You do not have to return the stack that was passed into the function.

Note: For the sorted stack, the top of the stack should have the greatest number

```
stack<int> sortStack(stack<int> st){



}
```

Stacks #2

```
stack<int> sortStack(stack<int> st){

    stack<int> s2;
    int n = 0;
    while(!st.empty()){
        n = st.top();
        st.pop();
        while(!s2.empty() && s2.top() > n){
            st.push(s2.top());
            s2.pop();
        }
        s2.push(n);
    }
    return s2;

}
```

Stacks #3

Write the following functions for the class stackAsArray which implements a stack using an array

```
class stackAsArray{
  private:
    int *stack;
    int max_size;
    int top;
  public:
    stackAsArray(int _max_size){
      stack = new int[_max_size];
      max_size = _max_size;
      top = -1; // -1 means stack is empty
    }
    void push(int);
    void pop();
};
```

Stacks #3

```
void stackAsArray::push(int x){
  if(top < max_size - 1){
    top++;
    stack[top] = x;
  }
}


void stackAsArray::pop(){
  if(top >= 0){
    top--;
  }
}
```

Queues #1

Match the following C++ queue functions.

a) Removes front of queue
b) Inserts into rear of queue
c) Returns size of queue
d) Returns front of queue
e) Returns whether queue is empty or not

```
void empty()  int size()  T front()  void pop()  void push(T)
```

Queues #1

Match the following C++ queue functions.

a) Removes front of queue - **`void pop()`**
b) Inserts into rear of queue - **`void push(T)`**
c) Returns size of queue - **`int size()`**
d) Returns front of queue - **`T front()`**
e) Returns whether queue is empty or not - **`bool empty()`**

`void empty()` **`int size()`** **`T front()`** **`void pop()`** **`void push(T)`**

Queues #2

Complete the function **`void separate(queue<int> &q, stack<int> &s)`** which receives a queue of unsorted integers and an empty stack. The function should separates the queue by removing all the odd numbers from the queue and putting them in the stack and leaves all the even numbers in the queue. Your function MUST have a space complexity of O(1), this means you cannot use any additional data structures (extra stack, extra queue, vector, string, array, etc.).

```
void separate(queue<int> &q, stack<int> &s){


}
```

Queues #2

```
void separate(queue<int> &q, stack<int> &s){
  int size = q.size();
  for(int i = 0; i < size; i++){
    if(q.front() % 2 != 0)
      s.push(q.front());
    else
      q.push(q.front());
    q.pop();
  }
}
```

Queues #3

Implement a queue using two stacks. The top of s1 should hold the front of the queue.

```
class QueueUsingStacks{
  private:
    stack<int> s1;
    stack<int> s2;
  public:
    QueueUsingStacks();
    void enqueue(int);
};
```

Queues #3

```
void QueueUsingStacks::enqueue(int x){
  while(!s1.empty()){
    s2.push(s1.top());
    s1.pop();
  }
  s1.push(x);
  while(!s2.empty()){
    s1.push(s2.top());
    s2.pop();
  }
};
```

Hashing #1

Insert the following values: {100, 18, 31, 30, 8, 11} into a hash table of size 10 using **linear probing**. hash function is given below.

```
int hash(int x){
    return x % 10;
}
```

0	1	2	3	4	5	6	7	8	9

Hashing #1

```
int hash(int x){
    return x % 10;
}
```

{100, 18, 31, 30, 8, 11}

0	1	2	3	4	5	6	7	8	9
100	31	30	11					18	8

Hashing #2

Write the hashing function which performs double hashing if a collision is found. Assume indices that hold -1 are empty. x is the value which you should insert into the table.

```
int hash1(int x, int tableSize){
    return x % tableSize;
}

int hash2(int x){
    return 31 - (x % 31);
}

void doubleHashing(int table[], int x, tableSize){



}
```

Hashing #2

```
int hash1(int x, int tableSize){
    return x % tableSize;
}

int hash2(int x){
    return 31 - (x % 31);
}

void doubleHashing(int table[] int x, tableSize){
    int index = 0;
    for(int i = 0; i < tableSize; i++){
        index = (hash1(x, tableSize)+ (i*hash2(x))) % tableSize;
        if(table[index] == -1){
            table[index] = x;
            break;
        }
    }
}
```

Sorting #1

Fill in the table below:

Sorting Algorithm	Best Case Time Complexity	Worst Case Time Complexity
MergeSort		
QuickSort		
HeapSort		
ShellSort		
BucketSort (Radix)		

Sorting #1

Fill in the table below:

Sorting Algorithm	Best Case Time Complexity	Worst Case Time Complexity
MergeSort	O(nlogn)	O(nlogn)
QuickSort	O(nlogn)	$O(n^2)$
HeapSort	O(nlogn)	O(nlogn)
ShellSort	O(nlogn)	$O(n(\log n)^2)$
BucketSort (Radix)	O(nk)	O(nk)

Sorting #2

Sort the array below using merge sort.

```
{5, 13, 56, 23, 2, 4, 75, 19}
```

Sorting #2

Sort the array below using merge sort.

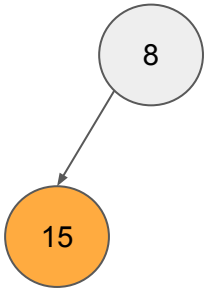
```
{5, 13, 56, 23, 4, 2, 75, 19}

{5, 13, 56, 23}    {4, 2, 75, 19}

{5, 13}  {56, 23}  {4, 2}  {75, 19}

{5} {13} {56} {23} {4} {2} {75} {19}

{5, 13}  {23, 56}  {2, 4}  {19, 75}

{5, 13, 23, 56}    {2, 4, 19, 75}

{2, 4, 5, 13, 19, 23, 56, 75}
```

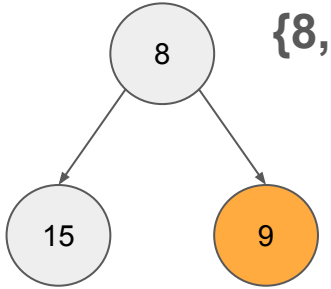
Heaps #1

Build a min-heap for the following array

{8, 15, 9, 3, 12, 5}

Show what the min-heap (either as a binary tree or array) looks like after every insertion.



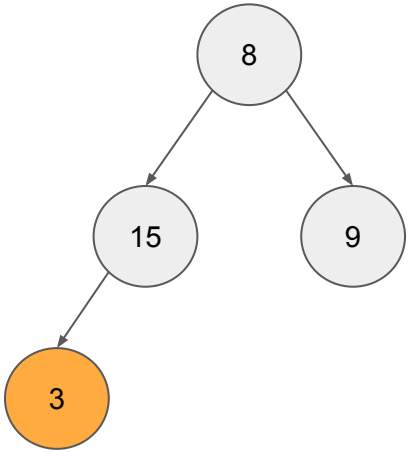


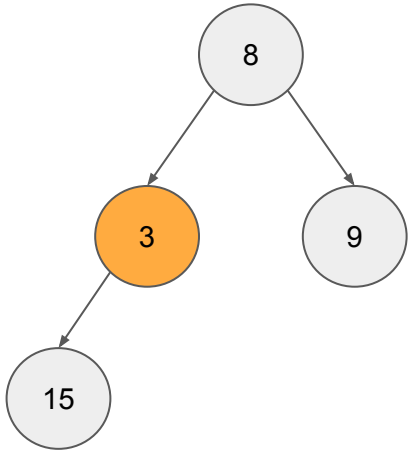
{8, 15, 9, 3, 12, 5}

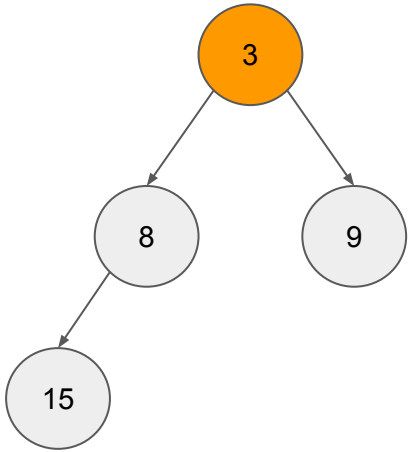
8					

8	15				

8	15	9			





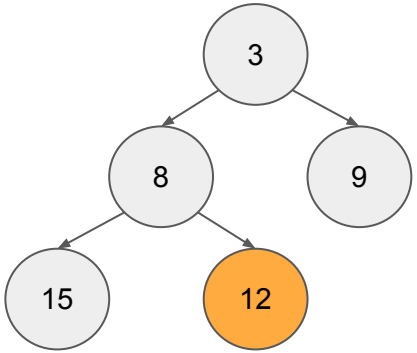


8	15	9	3		

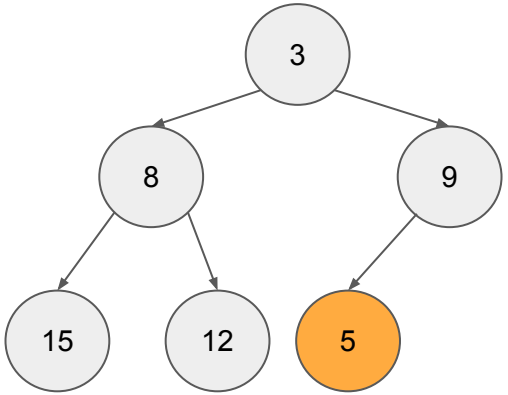
8	3	9	15		

3	8	9	15		

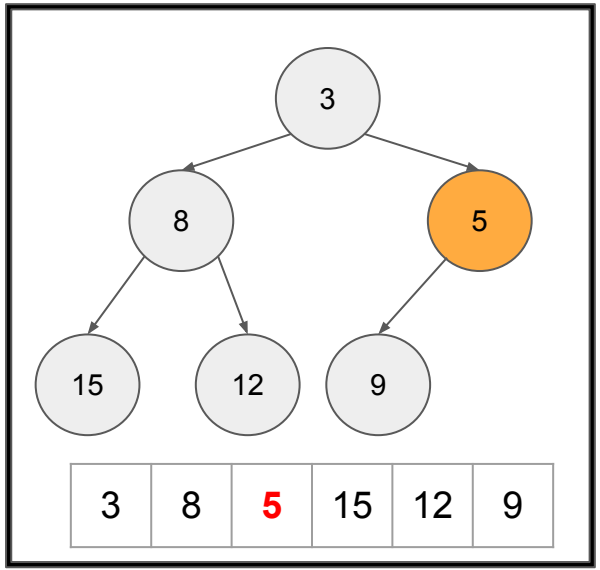
{8, 15, 9, 3, 12, 5}



3	8	9	15	12	



3	8	9	15	12	5



Recursion #1

Complete the **`void insertAtBottom(stack<int> &s, int x)`** which inserts the value **`x`** at the bottom of a stack using recursion.

```
void insertAtBottom(stack<int> &s, int x){




}
```

Recursion #1

```
void insertAtBottom(stack<int> &s, int x){
  if(s.empty())
    s.push(x);
  else{
    int tempVal = s.top();
    s.pop();
    insertAtBottom(s, x);
    s.push(tempVal);
  }
}
```