Exam 2 Review

Review Problems Covered in Workshop + More

Stacks

1. **Linked List Implementations**
   a. push()

```
// Push a value onto the stack
void push(int val) {
    Node* newItem = new Node;
    newItem->data = val;
    newItem->next = topNode;
    topNode = newItem;
}
```

   b. pop()

```
// Pop the top element from the stack
void pop() {
    if (!isEmpty()) {
        Node* temp = topNode;
        topNode = topNode->next;
        delete temp;
    } else {
        cout << "Stack is empty. Cannot pop." << endl;
    }
}
```

   c. top()

```
// Get the top element of the stack
int top() {
    if (!isEmpty()) {
        return topNode->data;
    }
    // Return a sentinel value or throw an exception for an empty stack
    cout << "Stack is empty. Returning sentinel value." << endl;
    return -1;
}
```

   d. isEmpty()

```
// Check if the stack is empty
bool isEmpty() {
    return topNode == nullptr;
}
```

2. **Array-Based Implementations**

a. push()

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

b. pop()

```
void StackAsArray::pop() {
    if (cur_size >= 0) {
        cur_size--;
    } else {
        cout << "Stack is empty. Cannot pop." << endl;
    }
}
```

c. top()

```
int StackAsArray::top() {
    if (cur_size >= 0) {
        return stack[cur_size];
    }
    // Handle the case where the stack is empty. You can return a default value or throw an exception.
    cout << "Stack is empty. Returning sentinel value." << endl;
    return 0;
}
```

d. isEmpty()

```
bool StackAsArray::isEmpty() {
    return cur_size == -1;
}
```

e. isFull()

```
bool isFull() {
    return cur_size == max_size - 1;
}
```

3. **Infix to Postfix (Implementation & Tracing)**

```
string infixToPostfix(string str)
{
    stack<char> s;
    string postfix = "";

    //iterate through string
    for(int i = 0; i < str.length(); i++)
    {
        //if character is an operand, add to postfix string
        if(isdigit(str[i]))
            postfix += str[i];

        //push opening parenthesis to stack
        else if(str[i] == '(')
            s.push(str[i]);

        //when you find a closing parenthesis, pop all operators from stack
        //until you reach the opening parenthesis
        else if(str[i] == ')')
        {
            while(s.top() != '(')
            {
                postfix += s.top();
                s.pop();
            }

            //discard opening parenthesis
            s.pop();
        }

        //if you find any other operator
        else
        {
            while(!s.empty() && checkOperatorPriority(str[i]) <= checkOperatorPriority(s.top()))
            {
                postfix += s.top();
                s.pop();
            }

            s.push(str[i]);
        }
    }

    while(!s.empty())
    {
        postfix += s.top();
        s.pop();
    }

    return postfix;
}
```

4. **Postfix to Infix (Implementation & Tracing)**

```
string postfixToInfix(const std::string& exp) {
    stack<std::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();
}
```

5. **Evaluate Postfix (Implementation & Tracing)**

```
int evaluatePostfix(string exp) {
    stack<int> s;

    for (int i = 0; i < exp.length(); i++) {
        if (isdigit(exp[i])) {
            s.push(exp[i] - '0');
        } else {
            int val1 = s.top();
            s.pop();
            int val2 = s.top();
            s.pop();

            switch (exp[i]) {
                case '+':
                    s.push(val2 + val1);
                    break;
                case '-':
                    s.push(val2 - val1);
                    break;
                case '*':
                    s.push(val2 * val1);
                    break;
                case '/':
                    s.push(val2 / val1);
                    break;
            }
        }
    }

    return s.top();
}
```

6. **Given an expression, find out whether or not that expression contains valid parentheses. An expression contains valid parenthesis if it has the proper parenthesis ( ), [ ], { } in the correct order.**

string = "{ [ ] (( )) }" → true
string = "{ ( ]) } }" → false

```
bool validParentheses(string exp) {
    stack<char> s;

    for (int i = 0; i < exp.length(); i++) {
        if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{') {
            s.push(exp[i]);
        } else if (exp[i] == ')') {
            if (s.empty() || s.top() != '(') {
                return false;
            }
            s.pop();
        } else if (exp[i] == ']') {
            if (s.empty() || s.top() != '[') {
                return false;
            }
            s.pop();
        } else if (exp[i] == '}') {
            if (s.empty() || s.top() != '{') {
                return false;
            }
            s.pop();
        }
    }

    return s.empty();
}
```

7. **Reverse a stack using recursion (you are only allowed access to the stack passed in through your function, no additional data structures allowed). Added printStack() as well.**

```
void reverseStack(stack<int> &s){
  if(s.empty()){
    return;
  }
  else{
    int top = s.top();
    s.pop();
    reverseStack(s);
    if(!s.empty()){
      int temp = s.top();
      s.pop();
      reverseStack(s);
      s.push(top);
      reverseStack(s);
      s.push(temp);
    }
    else{
      s.push(top);
    }
  }
}
```

```
void printStack(stack<int> s){
  while(!s.empty()){
    cout << s.top() << " ";
    s.pop();
  }
  cout << endl;
}
```

8. **Given a value, return the index of where it is in the stack (assume the top of the stack is index 0, and the bottom of the stack is index n - 1, n being the size of the stack). If the value doesn't exist in the stack, return -1 (iteratively and recursively).**

```
// Iterative version to find the index of a value in the stack.
int returnIndexIterative(stack<int> &s, int value) {
    stack<int> tempStack; // Temporary stack to hold the elements during the search.
    int index = 0;

    // Pop elements from the original stack and push them to the temporary stack
    // while checking for the value.
    while (!s.empty()) {
        if (s.top() == value) {
            // If the value is found, return the index.
            return index;
        }
        tempStack.push(s.top());
        s.pop();
        index++;
    }

    // If the value is not found in the stack, return -1.
    return -1;
}
```

```
// Recursive version to find the index of a value in the stack.
int returnIndexRecursive(stack<int> &s, int value, int index = 0) {
    // Base case: If the stack is empty, the value is not found, return -1.
    if (s.empty()) {
        return -1;
    }

    // Check the top element of the stack.
    if (s.top() == value) {
        // If the value is found, return the index.
        return index;
    }

    // Pop the top element and recursively search the rest of the stack.
    s.pop();
    return returnIndexRecursive(s, value, index + 1);
}
```

9. **Write the function stack<int> sortStack(stack<int> &st) which returns a sorted stack in descending order. You do not have to return the stack that was passed into the function. For the sorted stack, the top of the stack should have the greatest number.**

```
stack<int> sortStack(stack<int> st) {
    stack<int> s2;
    int n = 0;
    int size = st.size();

    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;
}
```

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

```
class StackAsArray {
private:
    int* stack;
    int max_size;
    int cur_size;

public:
    StackAsArray(int _max_size) {
        stack = new int[_max_size];
        max_size = _max_size;
        cur_size = -1; // -1 means the stack is empty
    }

    void push(int);
    void pop();
};

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

void StackAsArray::pop() {
    if (cur_size >= 0) {
        cur_size--;
    }
}
```

11. **Reverse a string using stacks**

```
string reverseString(string s){
  stack <char> st;

  // Push each character of the input string onto the stack
  for(int i = 0; i < s.length(); i++){
    st.push(s[i]);
  }

  // Pop characters from the stack and assign them back to the string to reverse it
  for(int i = 0; i < s.length(); i++){
    s[i] = st.top();
    st.pop();
  }

  return s;
}
```

12. ~~**Remove duplicates from a string using stacks**~~

```
string removeDuplicates(string s) {
  string ans;
  stack<char> st; // stack creation

  for (int i = 0; i < s.length(); i++) {
    if (st.empty()) { // if the stack is empty, push the current character
      st.push(s[i]);
    } else {
      if (st.top() ==
          s[i]) { // if the current character is the same as the top of the
                  // stack, pop it (remove adjacent duplicates)
        st.pop();
      } else {
        st.push(s[i]); // push the current character onto the stack
      }
    }
  }

  // other elements in the string
  while (!st.empty()) {
    ans += st.top();
    st.pop();
  }

  // reverse the string back
  reverse(ans.begin(), ans.end());

  return ans;
}
```

13. **Given a value, return the index of where it is in the stack (assume the top of the stack is index 0, and the bottom of the stack is index n - 1, n being the size of the stack). If the value doesn't exist in the stack, return -1.**

```
int searchStack(stack<int>&s, int targetNum){
  int index = 0;
  int maxSIze = s.size();

  while(!s.empty()){
    if(s.top() == targetNum){
      return index;
    }
    s.pop();
    index++;
  }
  return -1;
}
```

14. **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) {
    if (s.empty()) {
        s.push(x);
    } else {
        int tempVal = s.top();
        s.pop();
        insertAtBottom(s, x);
        s.push(tempVal);
    }
}
```

Queues

1. **Linked List Implementations**
   a. enqueue()

```
// Enqueue (add to the end)
void enqueue(int value) {
    Node* newNode = new Node(value);
    if (tail == nullptr) {
        head = newNode;
        tail = newNode;
    } else {
        tail->next = newNode;
        tail = newNode;
    }
}
```

   b. dequeue()

```
// Dequeue (remove from the front)
void dequeue() {
    if (head == nullptr) {
        cout << "The list is empty. Cannot dequeue." << endl;
    } else {
        Node* temp = head;
        head = head->next;
        delete temp;
        if (head == nullptr) {
            tail = nullptr; // Update tail if the list becomes empty
        }
    }
}
```

   c. front()

```
// Front (get the value at the front)
int front() {
    if (head == nullptr) {
        cout << "The list is empty. There is no front element." << endl;
        return -1; // You can choose a different value or approach for error handling
    } else {
        return head->data;
    }
}
```

   d. isEmpty()

```
// Check if the list is empty
bool isEmpty() {
    return head == nullptr;
}
```

2. **Array-Based Implementations**

a. enqueue()

```
// Enqueue function to add an element to the end of the queue
void enqueue(int item) {
    if (rearIndex == MAX_SIZE - 1) {
        cout << "Queue is full. Cannot enqueue." << endl;
        return;
    }

    if (isEmpty()) {
        frontIndex = 0; // If the queue is empty, set the front to 0
    }

    arr[++rearIndex] = item;
}
```

b. dequeue()

```
// Dequeue function to remove an element from the front of the queue
void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty. Cannot dequeue." << endl;
        return;
    }

    if (frontIndex == rearIndex) {
        // If there's only one element in the queue, reset the queue
        frontIndex = -1;
        rearIndex = -1;
    } else {
        frontIndex++;
    }
}
```

c. front()

```
// Front function to get the front element of the queue
int front() {
    if (isEmpty()) {
        cout << "Queue is empty." << endl;
        return -1; // You can choose to return a special value for an empty queue
    }
    return arr[frontIndex];
}
```

d. isEmpty()

```
// isEmpty function to check if the queue is empty
bool isEmpty() {
    return (frontIndex == -1 && rearIndex == -1);
}
```

3. **Priority Queues**

a. enqueue()

```
void enqueue(int value, int priority) {
    if (isEmpty()) {
        pq.push(value);
        return;
    }

    int n = pq.size();
    bool flag = false;

    for (int i = 0; i < n; i++) {
        if (priority > pq.front() && !flag) {
            pq.push(value);
            flag = true;
        }
        pq.push(pq.front());
        pq.pop();
    }

    if (!flag) {
        pq.push(value);
    }
}
```

b. dequeue()

```
// dequeue the highest priority
void dequeue() {
    if (!isEmpty()) {
        pq.pop();
    }
}
```

c. front()

```
// get the element with the highest priority
int front() {
    if (!isEmpty()) {
        return pq.front();
    } else {
        return -1;
    }
}
```

d. isEmpty()

```
// check if priority queue is empty
bool isEmpty() {
    return pq.empty();
}
```

4. **What are the differences between a queue and priority queue?**

A queue follows the "first-in, first-out" (FIFO) principle, where the first element added is the first to be removed. In contrast, a priority queue does not follow a strict order and removes elements based on their priority, allowing higher-priority elements to be removed first.

5. **Implement a function that takes in two arrays, one that holds a list of tasks, and another that holds each task's associated time to complete and performs round-robin scheduling on the list of tasks (quantum time is the amount of time to perform a task for before moving on to the next). Output the tasks in the order that they're completed.**

```
void roundRobinScheduling(vector<string>& tasks, vector<int>& completionTimes,
int quantumTime) {
    queue<string> taskQueue;
    queue<int> timeQueue;

    int n = tasks.size();

    // Initialize the queues with the tasks and their completion times
    for (int i = 0; i < n; i++) {
        taskQueue.push(tasks[i]);
        timeQueue.push(completionTimes[i]);
    }

    int currentTime = 0;

    while (!taskQueue.empty()) {
        string currentTask = taskQueue.front();
        int remainingTime = timeQueue.front();

        if (remainingTime <= quantumTime) {
            // The current task can be completed within the quantum time
            currentTime += remainingTime;
            taskQueue.pop();
            timeQueue.pop();
            cout << currentTask << " ";
        } else {
            // The current task still needs more time to complete
            currentTime += quantumTime;

            // Update the remaining time for the task
            timeQueue.front() -= quantumTime;

            // Rotate the task to the back of the queue
            taskQueue.push(taskQueue.front());
            timeQueue.push(timeQueue.front());

            taskQueue.pop();
            timeQueue.pop();
            cout << currentTask << " ";
        }
    }

    cout << endl;
}
```

6. **Implement a function that reverses a queue recursively (you're not allowed other data structures, only have access to the queue STL functions - push(), pop(), front(), size()...).**

```
void reverseQueue(queue<int>& q) {
    if (q.empty()) {
        return; // Base case: the queue is empty
    }

    int frontElement = q.front();
    q.pop();

    reverseQueue(q); // Recursively reverse the smaller queue

    q.push(frontElement); // Push the front element to the back of the reversed queue
}
```

7. **Implement a function that returns the number of even elements in a queue (you cannot use any additional data structures. Elements in the queue should as well be in the same order as when passed through the function after the function is done running).**

```
int countEven(queue<int>& q) {
    int evenCount = 0;
    int originalSize = q.size(); // Get the original size of the queue

    // Process the elements in the queue and count the even numbers
    for (int i = 0; i < originalSize; i++) {
        int frontElement = q.front();
        if (frontElement % 2 == 0) {
            evenCount++;
        }
        q.pop();
        q.push(frontElement); // Put the element back at the end to preserve order
    }

    // Restore the queue to its original order
    for (int i = 0; i < originalSize; i++) {
        int frontElement = q.front();
        q.pop();
        q.push(frontElement);
    }

    return evenCount;
}
```

8. **The function receives a queue of unsorted integers and an empty stack. The function should separate the queue by removing all the odd numbers from the queue and putting them in the stack and leaving 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) {
    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();
    }
}
```

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

```
class QueueUsingStacks {
private:
    std::stack<int> s1;
    std::stack<int> s2;

public:
    QueueUsingStacks();
    void enqueue(int);
};

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();
    }
}
```

10. **Write the function void levelOrder(int heap[], int size) which prints a heap in level order. For example: levelOrder([10, 9, 6, 5, 1, 2, 3], 7) will produce: 10 9 6 5 1 2 3.**

```
void levelOrder(int heap[], int size) {
    queue<int> q;
    int counter = 1;

    q.push(0);
    q.push(-1);

    while (!q.empty()) {
        int i = q.front();
        q.pop();

        if (i != -1) {
            cout << heap[i] << " ";

            if ((2 * i + 1) < size) {
                q.push(2 * i + 1);
                counter++;
            }

            if ((2 * i + 2) < size) {
                q.push(2 * i + 2);
                counter++;
            }
        } else {
            cout << endl;
            if (counter < size) {
                q.push(-1);
            }
        }
    }
}
```

11. **Write the function void roundRobin(process arr[], int n, int qt) which executes round robin scheduling on the processes in arr[]. n is the number of processes in arr[] and qt is the quantum time. Your function should print the processes in the order in which they are finished (this means their time has reached 0). Example: roundRobin([P1(5), P2(3), P3(9), P4(5)], 4, 3) will produce: P2 P1 P4 P3.**

```
struct Process {
    string ID;
    int time;
};

void roundRobin(Process arr[], int n, int qt) {
    queue<Process> q;
    Process curP;

    for (int i = 0; i < n; i++) {
        q.push(arr[i]);
    }

    while (!q.empty()) {
        curP = q.front();
        q.pop();

        curP.time = curP.time - qt;

        if (curP.time <= 0) {
            cout << curP.ID << " ";
        } else {
            q.push(curP);
        }
    }

    cout << endl;
}
```

12. **Implement a function rearrangeQueue(queue<int> q) that rearranges the given queue of integers. The rearrangement should place the even numbers at the front of the queue and the odd numbers at the back. The relative order of even and odd numbers should be maintained. You can use 2 single queues. You cannot use any additional data structures or recursion.**

```
void rearrangeQueue(queue<int> &q){
  queue<int> evenQ, oddQ;

  while(!q.empty()){
    int frontVal = q.front();
    q.pop();

    if(frontVal % 2 == 0){
      evenQ.push(q.front());
    }

    else{
      oddQ.push(q.front());
    }
  }

  while(!evenQ.empty()){
    q.push(evenQ.front());
    evenQ.pop();
  }

  while(!oddQ.empty()){
    q.push(oddQ.front());
    oddQ.pop();
  }
}
```

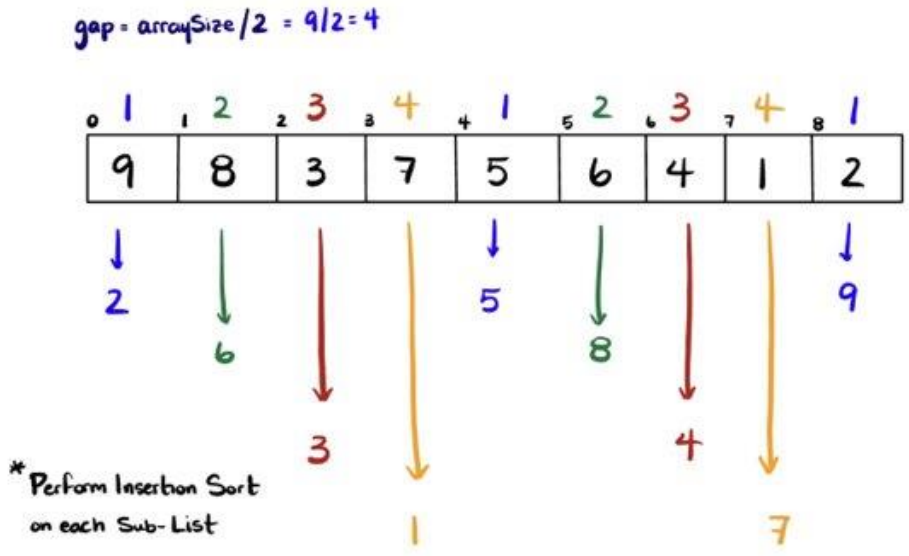
Sorting

1. **Merge, Quick, Shell, Bucket and Heap Sort (Implementations, Tracing and Time Complexities)**

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)

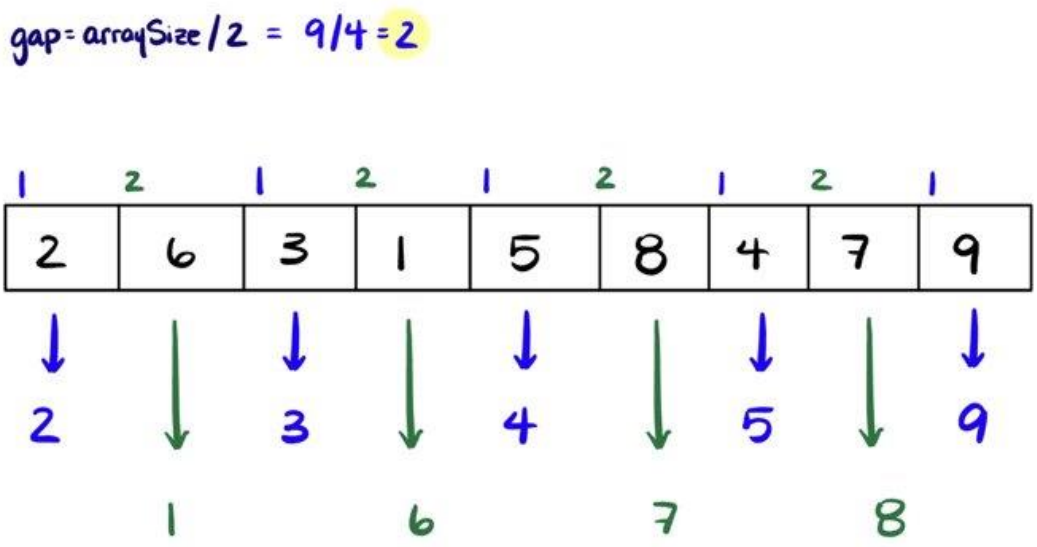
-Shell Sort

- 1st Iteration (Gap Size = 4, numbers represent their own respective sub-array)



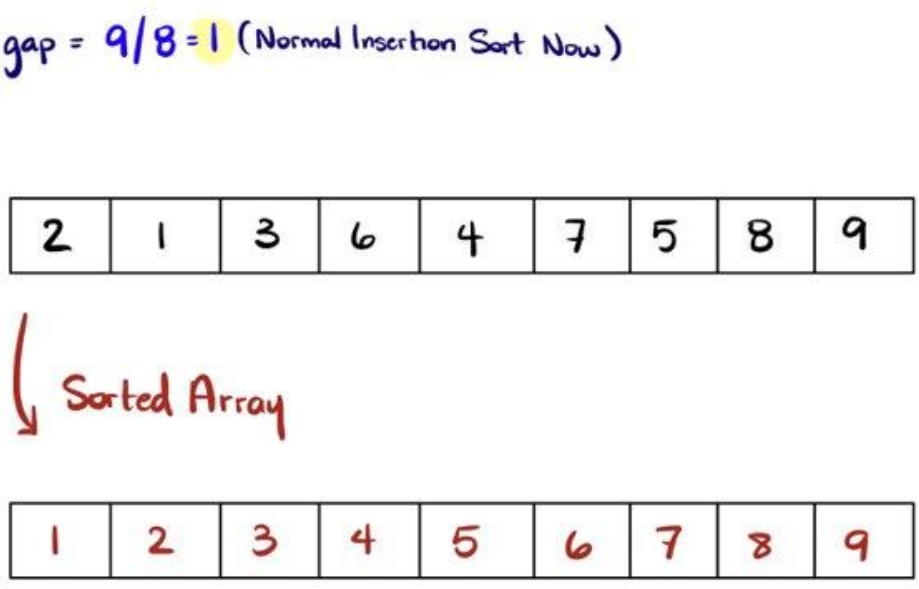
Here, we see the array is divided into different sublists (determined by gapValue). We then perform insertion sort on every sublist. New array is saved to continue to perform the shell sort with a smaller gap value.

- 2nd Iteration (Gap Size = 2)



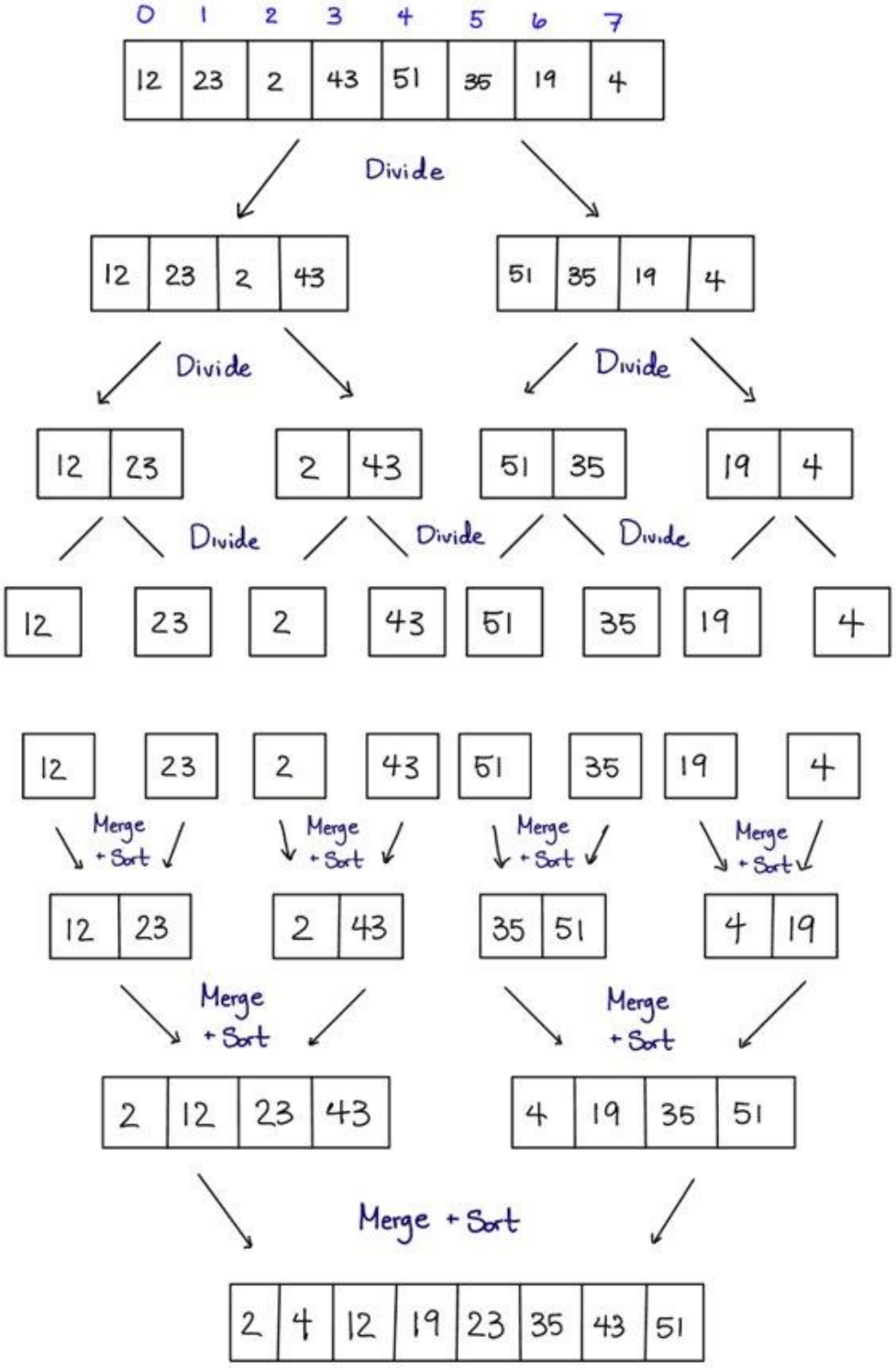
Shell sort with gap value of 2

- Last Iteration (Since Gap = 1, at this point, standard insertion sort proceeds)

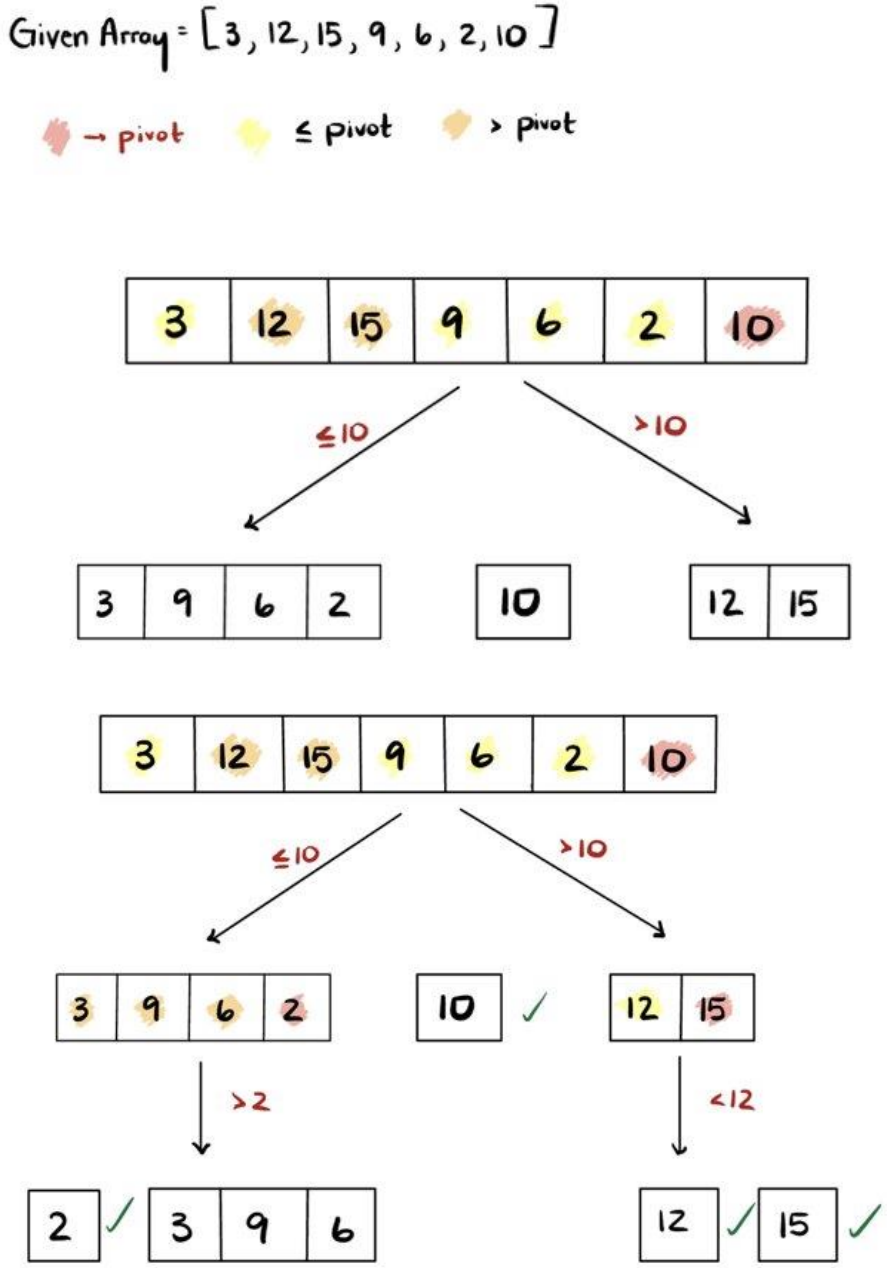


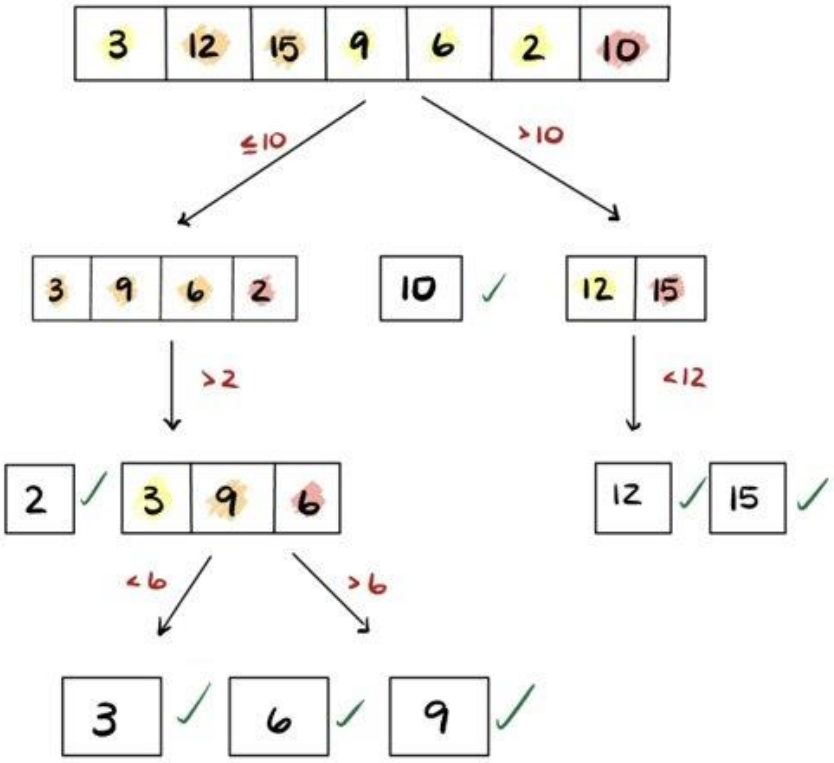
Last iteration of the shell sort, where the gap value is 1, where we're basically just doing insertion sort

-Merge Sort

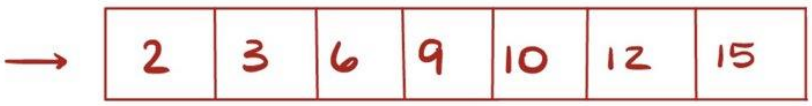

0 1 2 3 4 5 6 7
12 23 2 43 51 35 19 4
Divide
12 23 2 43
51 35 19 4
Divide
Divide
12 23
2 43
51 35
19 4
Divide
Divide
Divide
12
23
2
43
51
35
19
4
Merge + Sort
12 23
2 43
35 51
4 19
Merge + Sort
Merge + Sort
2 12 23 43
4 19 35 51
Merge + Sort
2 4 12 19 23 35 43 51

-Quick Sort


Given Array = [3, 12, 15, 9, 6, 2, 10]
→ pivot
≤ pivot
> pivot
3 12 15 9 6 2 10
≤10
>10
3 9 6 2
10
12 15
3 12 15 9 6 2 10
≤10
>10
3 9 6 2
10
12 15
>2
<12
2
3 9 6
12
15



Merge Subarrays at End

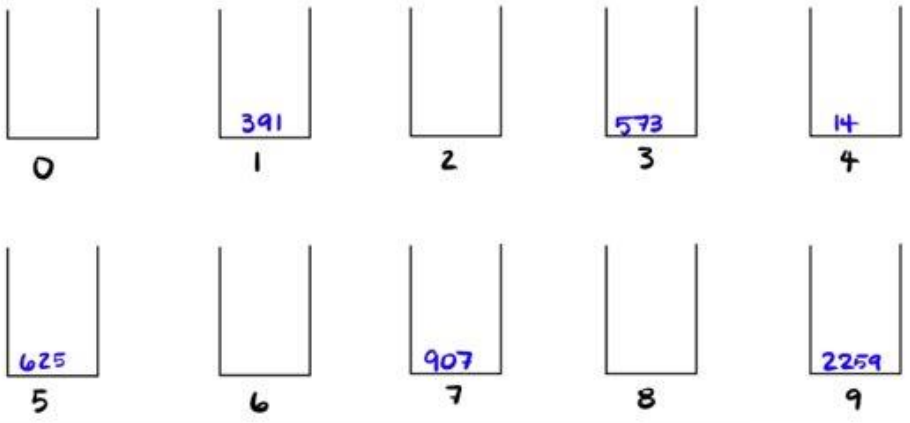


-Bucket Sort

Given Array = [391, 625, 907, 14, 2259, 573]

LSD: 391, 625, 907, 14, 2259, 573

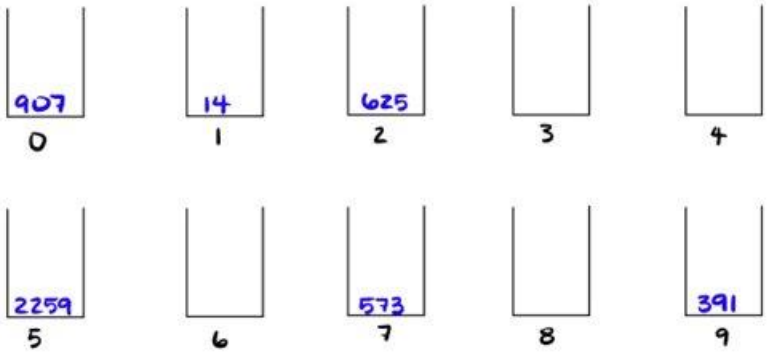
Place in respective bucket:



Empty Buckets in Order

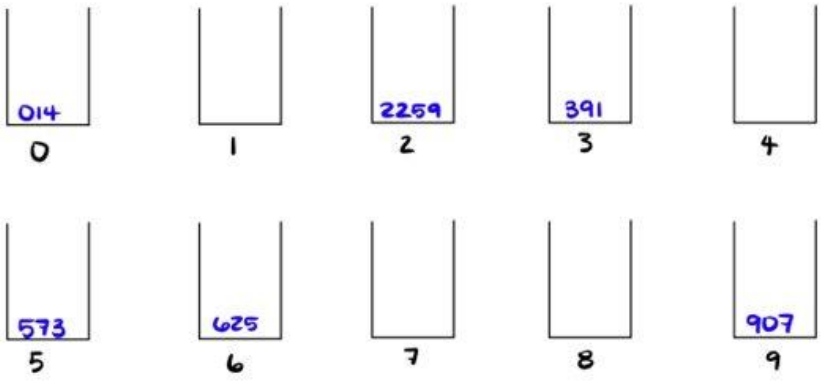
→ 391, 573, 14, 625, 907, 2259

Next LSD: 391, 573, 14, 625, 907, 2259



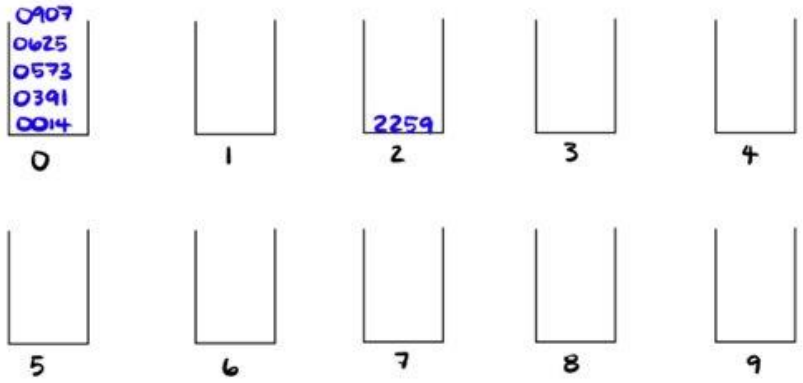
Empty Buckets → 907, 14, 625, 2259, 573, 391

Next LSD: 907, 014, 625, 2259, 573, 391



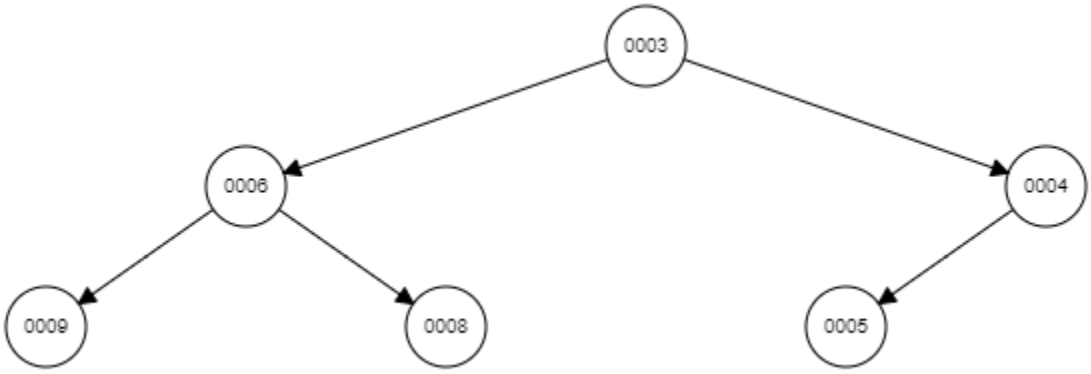
Empty Buckets → 0014, 2259, 0391, 0573, 0625, 0907

Greatest Significant Digit: 0014, 2259, 0391, 0573, 0625, 0907



Sorted Values → 14, 391, 573, 625, 907, 2259

2. **Build a Min Heap from the following array, [6, 3, 5, 9, 8, 4]. After doing so, perform Heap Sort to it. In order it's always M, L, R, L, L, R, R... (middle, left, right)**



Hashing

1. **Direct Hashing, Linear/Double Proving, Double Hashing and Separate Chaining (Implementations and Tracing).**

Base Code:

```
for (O -> N) {
    int index = SOMETHING;
    if (arr[index] == -1) {
        arr[index] = x;
        return;
    }
}
```

-Direct Hashing: Something = i % size

-Linear: Something = (hash(x) + i) % size

-Quadradic: Something = (hash(x) + i^2) % size

-Double Hashing: Something = (hash1(x) + i hash2(x)) % size

-Separate Chaining: Separate chaining is a collision resolution technique in hashing where each bucket in the hash table stores a linked list of key-value pairs, allowing multiple items with the same hash to be stored together. It provides a simple and efficient way to handle collisions in hash tables.

```
void separateChaining(node* table[], int val, int tableSize){
  int index = val % tableSize;
  node* temp = new node(val);

  if(table[index] == nullptr)
  {
      table[index] = temp;
  }

  else
{
    node* curr = table[index];

    while(curr -> next != nullptr)
    {
      curr = curr -> next;
    }

    curr -> next = temp;
  }
}
```

2. **Insert the following values into a Hash Table of size 5 using Double Hashing**

10, 20, 15, 30, 45*

0	1	2	3	4
10	20	15	30	45

x	i	h1(x)	h2(x)	i * h2(x)	h1(x) + (i * h2(x)) % n
10	0	0	2	0	0
20	0	0	2	0	0
20	1	0	2	2	2
15	0	0	2	0	0
15	1	0	2	2	2
15	2	0	2	4	4
30	0	0	3	0	0
30	1	0	3	3	3
30	2	0	3	6	1
45	0	0	1	0	0
45	1	0	1	1	1
45	2	0	1	2	2
45	3	0	1	3	3
45	4	0	1	4	4

Heaps

1. **Building Max/Min Heaps**

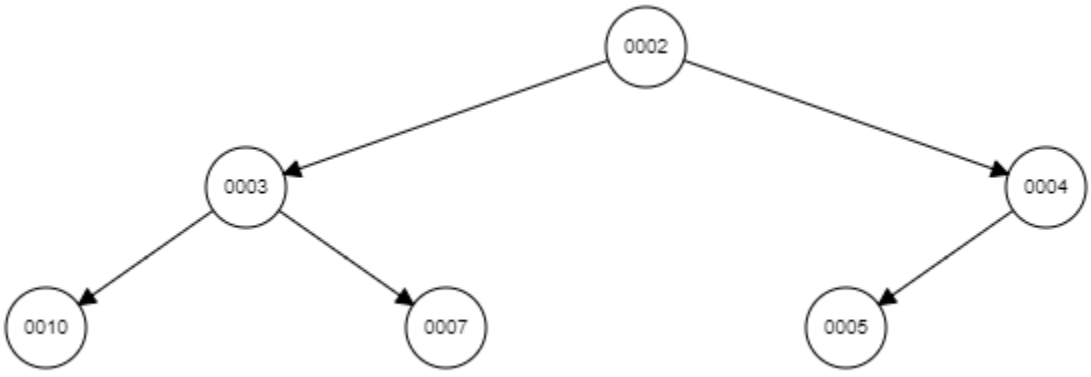
Max Heap:

- **In a Max Heap, for any given node, the value of the node is greater than or equal to the values of its children. The largest element is at the root.**
- **Start with an empty binary tree (array representation).**
- **Begin inserting elements from left to right into the tree. Insert the first element as the root.**
- **For each subsequent element, insert it as a child of the last inserted node and compare it with its parent. If the value of the newly inserted node is greater than its parent, swap the two nodes.**
- **Continue this process for all elements in the array until the entire array is organized into a Max Heap.**

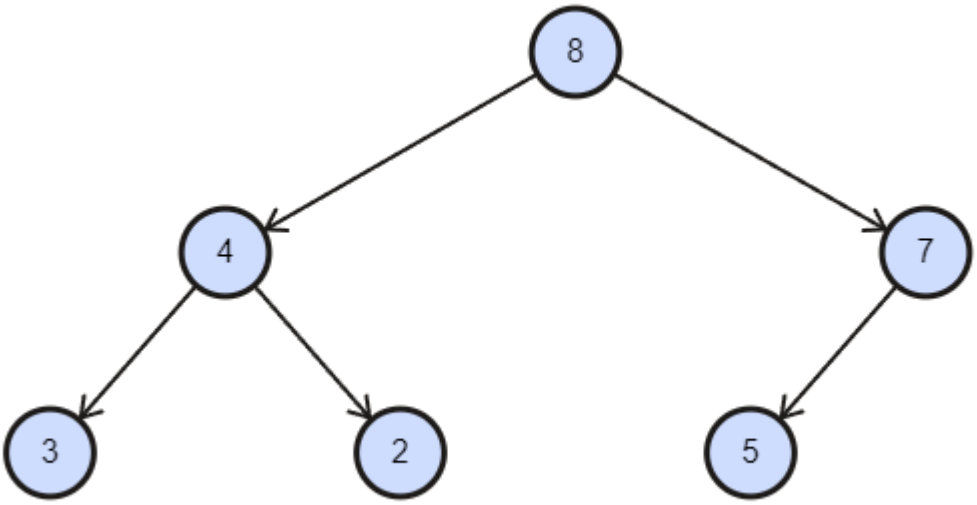
Min Heap:

- **In a Min Heap, for any given node, the value of the node is smaller than or equal to the values of its children. The smallest element is at the root.**
- **Start with an empty binary tree (array representation).**
- **Begin inserting elements from left to right into the tree. Insert the first element as the root.**
- **For each subsequent element, insert it as a child of the last inserted node and compare it with its parent. If the value of the newly inserted node is smaller than its parent, swap the two nodes.**
- **Continue this process for all elements in the array until the entire array is organized into a Min Heap.**

2. **Build a Min Heap from the following array, [10, 2, 5, 3, 7, 4]**



3. **Build a Max Heap from the following array, [2, 4, 7, 3, 8, 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) {
    if (s.empty()) {
        s.push(x);
    } else {
        int tempVal = s.top();
        s.pop();
        insertAtBottom(s, x);
        s.push(tempVal);
    }
}
```

2. **Basic Recursion Tree**

```
//IN ORDER - SEQUENTIAL
void recursiveFunction(int n) {
    if (n <= 0) {
        return;
    }
    recursiveFunction(n - 1); // Recursive call before cout
    cout << n << " ";
    //For recursiveFunction(5), it would print: "1 2 3 4 5".
}

// REVERSE ORDER - NON-SEQUENTIAL
void recursiveFunction2(int n) {
    if (n <= 0) {
        return;
    }
    cout << n << " ";
    recursiveFunction2(n - 1); // Recursive call after cout
    //For recursiveFunction(5), it would print: "5 4 3 2 1".
}
```

3. **Factorial**

factorial.cpp > factorial(int)

```
1   int factorial(int input){
2       if (input <=1)
3           return 1;
4       return input*factorial(input-1);
5   }
```

4. **Fibonacci**

```
int fib(int num){
  if(num <= 1 ){
    return num;
  }

 return fib(num -1) + fib(num-2);
}
```

5. **Recursive Sum**

```
int recurSum(int n)
{
    if (n <= 1)
        return n;
    return n + recurSum(n - 1);
}
```

6. **Palindrome**

```
bool Palindrome(string s, int start, int end){
  //edge case
  if(s.length() <= 2){
    return true;
  }
  //base case
  if(start >= end){
    return true;
  }else if(s[start] == s[end]){
    //recursive case
    return Palindrome(s, start+1, end-1);
  }else{
    return false;
  }
}
```

7. **ReturnIndex**

```
// Recursive version to find the index of a value in the stack.
int returnIndexRecursive(stack<int> &s, int value, int index = 0) {
    // Base case: If the stack is empty, the value is not found, return -1.
    if (s.empty()) {
        return -1;
    }

    // Check the top element of the stack.
    if (s.top() == value) {
        // If the value is found, return the index.
        return index;
    }

    // Pop the top element and recursively search the rest of the stack.
    s.pop();
    return returnIndexRecursive(s, value, index + 1);
}
```

8. **Reverse Stack**

```
void reverseStack(stack<int> &s){
  if(s.empty()){
    return;
  }
  else{
    int top = s.top();
    s.pop();
    reverseStack(s);
    if(!s.empty()){
      int temp = s.top();
      s.pop();
      reverseStack(s);
      s.push(top);
      reverseStack(s);
      s.push(temp);
    }
    else{
      s.push(top);
    }
  }
}
```

9. **Reverse Queue**

```
void reverseQueue(queue<int>& q) {
    if (q.empty()) {
        return; // Base case: the queue is empty
    }

    int frontElement = q.front();
    q.pop();

    reverseQueue(q); // Recursively reverse the smaller queue

    q.push(frontElement); // Push the front element to the back of the reversed queue
}
```