National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

School of Electrical Engineering and Computer Science

CS250 – Data Structures and Algorithms

Assignment 3: Sorting Algorithms and Heap Data Structure

Submission (Group) Details

Names	**CMS ID**
Muhammad Umer	345834
Muhammad Ahmed Mohsin	333060
Group	GP – 1
Instructor	Bostan Khan
Class	BEE12
Date	05/05/2024

1 Table of Contents

2 Sorting Algorithms and Heap Data Structure

2.1 Objective

The objective of this assignment is to familiarize students with various sorting algorithms, their implementations, complexities, and applications. The students will also implement priority queues using the heap data structure.

Please read the instructions for each task with great care and be sure to implement the requirements stated against each task.

This is a group assignment where each group can have a maximum of 2 students. Remember to include the names and registration numbers of both students at the start of the report.

Remember to document your work well in the documentation report. Explain the code, its design and test results with details for obtaining max marks for the whole assignment.

2.2 Deliverables

You will have to submit a separate .cpp source file for each task. Present the documentation report including the test results in a well-prepared PDF.

The report PDF and the sources files should be compressed in a single zip file and submitted on LMS with the name in following format: <Student_names>_<Reg_no>.zip

2.3 Documentation Report PDF

Provide a report PDF for all your linked list implementations, including explanations of the data structures and algorithms used, as well as any design decisions made for each task. Document the usage of your linked list in each of the application scenarios, including how the linked list data structure is utilized to solve the problem. **The marks of your coding tasks will be directly influenced by the documentation report.** Include clear instructions on how to compile/run your code and how to interact with the implemented applications.

2.4 Additional Notes:

- You are encouraged to be creative and innovative in your implementations. Consider additional features or optimizations that could enhance the functionality or performance of your applications.
- Collaboration with classmates is allowed for discussing concepts and problem-solving strategies, but each student must submit their own individual solution.
- **Plagiarism or copying of code from other students will result in zero marks.**

3 Tasks

3.1 Task 1: Research [20]

Research and provide brief explanations for the following sorting algorithms:

Bubble Sort
Bubble sort is a simple sorting algorithm that repeatedly iterates through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process continues until the entire list is sorted. While easy to understand and implement, bubble sort is inefficient for large lists due to its quadratic time complexity.

Pseudocode

```
procedure bubbleSort(list)
  n = length(list)
  for i = 0 to n-2 do
    for j = 0 to n-i-2 do
      if list[j] > list[j+1] then
        swap(list[j], list[j+1])
      end if
    end for
  end for
end procedure
```

Time Complexities

- **Average:** $O(n^2)$
- **Best:** $O(n)$; Occurs if the list is already sorted
- **Worst:** $O(n^2)$

Insertion Sort
Insertion sort builds the final sorted array one element at a time. It iterates through the list, taking each element and inserting it into its correct position in the sorted portion of the list. This algorithm is efficient for small lists or partially sorted lists but still has a quadratic worst-case time complexity.

Pseudocode

```
procedure insertionSort(list)
  for i = 1 to length(list)-1 do
    key = list[i]
    j = i - 1
    while j >= 0 and list[j] > key do
      list[j+1] = list[j]
      j = j - 1
    end while
    list[j+1] = key
  end for
end procedure
```

Time Complexities

- **Average:** $O(n^2)$
- **Best:** $O(n)$; Occurs if the list is already sorted
- **Worst:** $O(n^2)$

Merge Sort

Merge sort is a divide-and-conquer algorithm that recursively divides the list into smaller sub-lists until each sub-list contains only one element (a list of one element is considered sorted). Then, it repeatedly merges the sub-lists back together, ensuring they are merged in sorted order, until the entire list is sorted. Merge sort has a time complexity of $O(n \log n)$ making it efficient for larger lists.

Pseudocode

```
procedure mergeSort(list)
  if length(list) > 1 then
    mid = length(list) / 2
    leftHalf = list[0...mid-1]
    rightHalf = list[mid...length(list)-1]
    mergeSort(leftHalf)
    mergeSort(rightHalf)
    merge(leftHalf, rightHalf, list)
  end if
end procedure

procedure merge(left, right, list)
  i = 0, j = 0, k = 0
  while i < length(left) and j < length(right) do
    if left[i] <= right[j] then
      list[k] = left[i]
      i = i + 1
    else
      list[k] = right[j]
      j = j + 1
    end if
    k = k + 1
  end while
  // Copy remaining elements of left, if any
  while i < length(left) do
    list[k] = left[i]
    i = i + 1
    k = k + 1
  end while
  // Copy remaining elements of right, if any
  while j < length(right) do
    list[k] = right[j]
    j = j + 1
    k = k + 1
  end while
end procedure
```

Time Complexities

- **Average:** $O(n \log n)$

- **Best:** $O(n \log n)$
- **Worst:** $O(n \log n)$

Heap Sort

Heap sort uses a heap data structure to efficiently sort elements. It involves building a max heap (or min heap for descending order) from the list, then repeatedly extracting the root element (the maximum element in a max heap), placing it at the end of the sorted list, and rebuilding the heap. This process continues until the entire list is sorted. Like merge sort, heap sort has a time complexity of $O(n \log n)$, making it suitable for large datasets.

Pseudocode

```
procedure heapSort(list)
  buildMaxHeap(list)
  for i = length(list) downto 2 do
    swap(list[1], list[i])
    heapify(list, i-1, 1)  // Note: heap size is now i-1
  end for
end procedure

procedure buildMaxHeap(list)
  for i = floor(length(list)/2) downto 1 do
    heapify(list, length(list), i)
  end for
end procedure

procedure heapify(list, n, i)
  largest = i
  left = 2*i
  right = 2*i + 1
  if left <= n and list[left] > list[largest] then
    largest = left
  end if
  if right <= n and list[right] > list[largest] then
    largest = right
  end if
  if largest != i then
    swap(list[i], list[largest])
    heapify(list, n, largest)
  end if
end procedure
```

Time Complexities

- **Average:** $O(n \log n)$
- **Best:** $O(n \log n)$
- **Worst:** $O(n \log n)$

3.2 Task 2: Implementation [30]

Implement merge sort and heap sort algorithms in C++. Provide well-commented code along with explanations of your implementation choices.

You should show the application of your merge sort and heap sort implementations on 3 examples each.

`Implementation Details: Merge Sort`

`merge``(arr, first, mid, last)`

- The function creates two temporary arrays, `L` and `R`, to store the left and right halves of the subarray being merged. Elements from the original array `arr` are copied into the temporary arrays `L` and `R` based on the calculated mid-point. The core of the function lies in the while loop. It compares elements from `L` and `R` and places the smaller element into the original array `arr` in sorted order. This process continues until one of the temporary arrays is exhausted.
- After one of the temporary arrays is empty, the remaining elements from the other array are directly copied into arr as they are already sorted.

`merge_sort``(arr, first, last)`

- Implements the divide-and-conquer strategy. It recursively divides the array into halves until each subarray contains only one element (base case for sorting).

`Implementation Details: Heap Sort`

`heapify``(arr, n, i)`

Note: The implementation assumes a `0-based indexing` for the array to ensure a fair comparison with merge sort in the subsequent task despite we, the students, learning `starting-from-1` indexing of arrays in class for heap sort.

- It assumes a max-heap structure and aims to maintain the heap property. The largest variable keeps track of the index of the largest element among the current node and its children. The indices of the left and right children are calculated as `2*i + 1` and `2*i + 2` respectively, based on the properties of a binary heap represented in an array.

`heap_sort``(arr, n)`

- The first loop iterates over non-leaf nodes of the array and calls `heapify` on each to build a max heap from the initial array.
- The second loop repeatedly extracts the root element `(largest element)` from the heap, swaps it with the last element in the unsorted region of the array, reduces the heap size, and calls `heapify` to maintain the heap property on the remaining elements.

Code

```
#include <iostream>
using namespace std;

// Recursive merge_sort and merge functions
void merge(int arr[], int first, int mid, int last) {
    int n1 = mid - first + 1;  // elements in the first half
    int n2 = last - mid;       // elements in the second half
    int L[n1], R[n2];          // temporary arrays

    // copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++) {
        L[i] = arr[first + i];
    }
    for (int j = 0; j < n2; j++) {
        R[j] = arr[mid + 1 + j];
    }

    // merge the temporary arrays back into arr[first..last]
    int i = 0, j = 0, k = first;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }

    // copy the remaining elements, if any
    // only one of the following two while loops will execute
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}

void merge_sort(int arr[], int first, int last) {
    if (first < last) {
        int mid = (first + last) / 2;
        merge_sort(arr, first, mid);
        merge_sort(arr, mid + 1, last);
        merge(arr, first, mid, last);
    }
}

// Heap sort functions (assuming index starts from 0)
void heapify(int arr[], int n, int i) {
    int largest = i;     // initialize largest as root
    int l = 2 * i + 1;   // left = 2*i + 1
    int r = 2 * i + 2;   // right = 2*i + 2

    if (l < n && arr[l] > arr[largest]) {
        largest = l;
    }

    if (r < n && arr[r] > arr[largest]) {
        largest = r;
    }

    // swap and continue heapifying if root is not largest
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void heap_sort(int arr[], int n) {
    // build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // one by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        // move current root to end
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Testing

The following `main` function provides the output for the required deliverable of `Task 2`; the outputs of both Heap and Merge sort on three distinct examples.

```
int main() {
    // Example 1
    int arr_merge[] = {12, 11, 13, 5, 6, 7};
    int arr_heap[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr_merge) / sizeof(arr_merge[0]);

    cout << "\nExample 1" << endl;
    cout << "=========" << endl;
    cout << "Merge Sort\n";
    merge_sort(arr_merge, 0, n - 1);
    for (int i = 0; i < n; i++) cout << arr_merge[i] << " ";
    cout << endl;

    cout << "Heap Sort\n";
    heap_sort(arr_heap, n);
    for (int i = 0; i < n; i++) cout << arr_heap[i] << " ";
    cout << endl;

    // Example 2
    int arr_merge2[] = {38, 27, 43, 3, 9, 82, 10};
    int arr_heap2[] = {38, 27, 43, 3, 9, 82, 10};
    n = sizeof(arr_merge2) / sizeof(arr_merge2[0]);

    cout << "\nExample 2" << endl;
    cout << "=========" << endl;
    cout << "Merge Sort\n";
    merge_sort(arr_merge2, 0, n - 1);
    for (int i = 0; i < n; i++) cout << arr_merge2[i] << " ";
    cout << endl;

    cout << "Heap Sort\n";
    heap_sort(arr_heap2, n);
    for (int i = 0; i < n; i++) cout << arr_heap2[i] << " ";
    cout << endl;

    // Example 3
    int arr_merge3[] = {5, 2, 4, 6, 1, 3};
    int arr_heap3[] = {5, 2, 4, 6, 1, 3};
    n = sizeof(arr_merge3) / sizeof(arr_merge3[0]);

    cout << "\nExample 3" << endl;
    cout << "=========" << endl;
    cout << "Merge Sort\n";
    merge_sort(arr_merge3, 0, n - 1);
    for (int i = 0; i < n; i++) cout << arr_merge3[i] << " ";
    cout << endl;

    cout << "Heap Sort\n";
    heap_sort(arr_heap3, n);
    for (int i = 0; i < n; i++) cout << arr_heap3[i] << " ";
    cout << endl;

    return 0;
}
```

Output

```
Example 1
=========
Merge Sort
5 6 7 11 12 13
Heap Sort
5 6 7 11 12 13

Example 2
=========
Merge Sort
3 9 10 27 38 43 82
Heap Sort
3 9 10 27 38 43 82

Example 3
=========
Merge Sort
1 2 3 4 5 6
Heap Sort
1 2 3 4 5 6
```

Instructions to Reproduce Output

```
                         Linux (Ubuntu 22.04)
- Compiler Version
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)

- Steps
cd to extracted .zip directory
user@hostname:/dsa/assignment_3# g++ -o task_2 task_2.cpp
user@hostname:/dsa/assignment_3# ./task_2

                              Windows
- Compiler Version
gcc version 13.2.0 (Rev3, Built by MSYS2 project)

- Steps
cd to extracted .zip directory
C:\user\dsa\assignment_1> g++ -Wall -g task_2.cpp -o task_2.exe
```

3.3 Task 3: Analysis of Sorting Algorithms [20]

1. Compare the time complexity of the two sorting algorithms you implemented.
2. Discuss the best-case, average-case, and worst-case scenarios for merge sort and heap sort algorithms.
3. Analyze and explain the advantages and disadvantages of each algorithm.

Comparison

```cpp
int main() {
    int n = 10000;
    int arr_best[n], arr_avg[n], arr_worst[n];
    for (int i = 0; i < n; i++) {
        arr_best[i] = i;
        arr_avg[i] = rand() % n;
        arr_worst[i] = n - i;
    }

    int arr_best_c[n], arr_avg_c[n], arr_worst_c[n];  // copy of arrays
    for (int i = 0; i < n; i++) {
        arr_best_c[i] = arr_best[i];
        arr_avg_c[i] = arr_avg[i];
        arr_worst_c[i] = arr_worst[i];
    }

    clock_t start, end;
    double time_taken;

    // merge sort
    start = clock();
    merge_sort(arr_best, 0, n - 1);
    end = clock();
    time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Merge sort best-case time: " << time_taken << " seconds\n";
    start = clock();
    merge_sort(arr_avg, 0, n - 1);
    end = clock();
    time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Merge sort average-case time: " << time_taken << " seconds\n";

    start = clock();
    merge_sort(arr_worst, 0, n - 1);
    end = clock();
    time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Merge sort worst-case time: " << time_taken << " seconds\n";

    // heap sort
    start = clock();
    heap_sort(arr_best_c, n);
    end = clock();
    time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "\nHeap sort best-case time: " << time_taken << " seconds\n";

    start = clock();
    heap_sort(arr_avg_c, n);
    end = clock();
    time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Heap sort average-case time: " << time_taken << " seconds\n";

    start = clock();
    heap_sort(arr_worst_c, n);
    end = clock();
    time_taken = double(end - start) / double(CLOCKS_PER_SEC);
    cout << "Heap sort worst-case time: " << time_taken << " seconds\n";

    return 0;
}
```

Output

```
Merge sort best-case time: 0.000598 seconds
Merge sort average-case time: 0.001065 seconds
Merge sort worst-case time: 0.000579 seconds

Heap sort best-case time: 0.001489 seconds
Heap sort average-case time: 0.001706 seconds
Heap sort worst-case time: 0.001506 seconds
```

Explanation

Merge sort has a *consistent* time complexity of $O(n \log n)$ for all cases (best, average, and worst). The algorithm always divides the array into halves and merges them back, leading to a logarithmic number of divisions and merges.

Like merge sort, heap sort also has a time complexity of $O(n \log n)$ for all cases. Building the heap takes $O(n)$ time and extracting each element from the heap takes $O(\log n)$ time, resulting in the overall complexity.

Observation

While both algorithms have the same theoretical time complexity, the observed measurements suggest that merge sort performed slightly better in this specific implementation and testing scenario. The actual performance can always vary depending on factors like the input data, hardware, and implementation details.

Analysis

merge_sort(int arr[], ...)	**heap_sort**(int arr[], ...)
Advantages	**Advantages**
○ **Stable** sorting algorithm (maintains the relative order of equal elements). ○ Can be easily **parallelized** due to its divide-and-conquer nature.	○ In-place sorting algorithm that incurs **minimal additional** memory. ○ **Can be faster** than merge sort in some cases when memory constrained.
Disadvantages	**Disadvantages**
○ **Requires additional memory** space for temporary arrays during the merging process.	○ Not a **stable** sorting algorithm, i.e., may change the relative order of equal elements.

3.4 Task 4: Heap Data Structure [30]

Implement a max-priority queue using the max-heap data structure in C++.
Include functions/methods for the following operations:

1. Inserting an element with a priority.
2. Extracting the maximum priority element.
3. Displaying the elements in the priority queue.
4. Increasing the priority of an element in the queue.
5. Discuss some applications of the implemented priority queue using the heap data structure.

Implementation

```
#include <climits>
#include <iostream>
using namespace std;

class MaxHeap {
   private:
    int *heap;
    int size;
    int capacity;

   public:
    MaxHeap(int total_capacity) {
        capacity = total_capacity;
        size = 0;
        heap = new int[capacity + 1];
        heap[0] = INT_MAX;  // dummy value
    }

    // Helper functions
    int parent(int i) {
        return i / 2;
    }
    int left_child(int i) {
        return 2 * i;
    }
    int right_child(int i) {
        return 2 * i + 1;
    }

    ~MaxHeap() {
        delete[] heap;
    }

    // heapify the subtree rooted at i
    void max_heapify(int i) {
        int left = left_child(i);
        int right = right_child(i);
        int largest = i;

        if (left <= size && heap[left] > heap[i]) {
            largest = left;
        }
        if (right <= size && heap[right] > heap[largest]) {
            largest = right;
        }

        if (largest != i) {
            swap(heap[i], heap[largest]);
            max_heapify(largest);
        }
    }

    // insert an element with a priority
    void insert(int key) {
        if (size == capacity) {
            cout << "Overflow: Heap is full\n";
            return;
        }

        size++;
        heap[size] = key;
        int i = size;
        while (i > 1 && heap[i] > heap[parent(i)]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }

    // extract the maximum priority element
    int extract_max() {
        if (size == 0) {
            cout << "Underflow: Heap is empty\n";
            return INT_MIN;
        }

        int max = heap[1];
        heap[1] = heap[size];
        size--;

        max_heapify(1);

        return max;
    }

    // display the elements in the priority queue
    void display() {
        if (size == 0) {
            cout << "Heap is empty\n";
            return;
        }

        cout << "\n** Priority Queue: ";
        for (int i = 1; i <= size; i++) {
            cout << heap[i] << " ";
        }
        cout << endl;
    }

    // increase the priority of an element in the queue
    void increase_priority(int i, int new_key) {
        if (i > size) {
            cout << "Invalid index\n";
            return;
        }

        if (new_key < heap[i]) {
            cout << "New key is smaller than the current key\n";
            return;
        }

        heap[i] = new_key;

        while (i > 1 && heap[i] > heap[parent(i)]) {
            swap(heap[i], heap[parent(i)]);
            i = parent(i);
        }
    }
};
```

Output

```
Max-Heap array:
** Priority Queue: 45 5 15 2 4 3

Extracted max: 45

Max-Heap array after extracting max:
** Priority Queue: 15 5 3 2 4

Max-Heap array after increasing key:
** Priority Queue: 20 15 3 2 4
```

Instructions to Reproduce Output

```
                              Linux (Ubuntu 22.04)
- Compiler Version
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)

- Steps
cd to extracted .zip directory
user@hostname:/dsa/assignment_3# g++ -o task_4 task_4.cpp
user@hostname:/dsa/assignment_3# ./task_2

                                    Windows
- Compiler Version
gcc version 13.2.0 (Rev3, Built by MSYS2 project)

- Steps
cd to extracted .zip directory
C:\user\dsa\assignment_1> g++ -Wall -g task_4.cpp -o task_4.exe
```

Applications/Use Cases

- **Network Simulation:** Simulating network traffic where packets have different priorities can utilize a priority queue to ensure high-priority packets are handled first.
- **Job Scheduling:** In operating systems, a priority queue can schedule tasks based on their priorities, ensuring that high-priority tasks get CPU time first.
- **Huffman Coding:** Priority queues play a crucial role in building the Huffman tree, which is used for data compression by assigning shorter codes to more frequent characters.
- **Hospital Emergency Room:** In a hospital emergency room, patients arrive with varying levels of urgency for medical attention; a priority queue can be used to manage patient triage.
- ... so on