**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**SCHOOL OF ELECTRONICS AND TELECOMMUNICATIONS**

**---🙠**🕮**🙢---**



**DATA STRUCTURE AND ALGORITHMS**

**PROJECT REPORT**

**Lecturer: Đào Trung Kiên**

**Course Code: ET2105E**

**Class Code: 236289**

**Semester: 2022.1**

**Group: 4**

**PROJECT: Implement and evaluate the Insertion, Bubble, Selection, Merge and Quick sort algorithms for doubly linked lists with generic types.**

**GROUP MEMBERS**

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ tên** | **Lớp** | **MSSV** |
| 1 | Dương Nhật Nam | CTTT Điện tử 01 K65 | 20203803 |
| 2 | Nguyễn Phạm Anh Tuấn | CTTT Điện tử 01 K65 | 20203827 |

TABLE OF WORK RESUME:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| No | Name | Student ID | Role | Work assignment | Completion |
| 1 | Dương Nhật Nam | 20203803 | Leader | * Design and code for classes. * Code for Bubble sort, Insertion sort, Merge sort and Quick sort * Write report. | Done |
| 2 | Nguyễn Phạm Anh Tuấn | 20203827 | Member | * Code for Selection sort. * Write report for Selection sort * Testing | Done |

1. **Introduction**

The aim of this project was to implement and evaluate 5 sorting algorithms, namely Insertion, Bubble, Selection, Merge and Quick sort, for doubly-linked lists with generic types in the C++ programming language.

1. **Implementation**

The five sorting algorithms were implemented using for loops and the results were evaluated in terms of time complexity and memory usage. The doubly-linked list data structure was created using a template class, where each node in the list has a data value and pointers to the previous and next nodes.

1. Class Node

* The code written to declare a Node:

Text

Description automatically generated

Figure 1. Class Node

* The class is defined using template class T.
* 3 public objects:

+ Variable ‘data’ has type T. It stores the value of each Node.

+ Pointer ‘next’ has type Node<T> stores address of the next Node in list.

+ Pointer ‘prev’ has type Node<T> stores address of the previous Node in list.

1. Class DoublyLinkedList

* The code written to declare a DoublyLinkedList:

A picture containing text, screenshot, software

Description automatically generated

Figure 2. Class DoublyLinkedList

* 2 public objects:

+ Pointer ‘head’ points to the first Node of the list.

+ Pointer ‘tail’ points to the last Node of the list.

* Constructor ‘DoublyLinkedList()’ initializes a list with head and tail pointers are nullptr.
* Method ‘Push(T n)’ appends an element at the end of the list and makes the tail pointer point to it. If the list has no element, this method adds the first element to the list.
* ‘SizeOfLL()’ returns an integer that is the length of the Linked List.
* ‘insertVector()’ takes a vector with type T and creates a doubly linked list with the vector’s elements.
* Method ‘Display()’ prints all the element in the list.
* ‘BubbleSort()’ is the method implemented to sort the linked list with Bubble sort algorithm.
* ‘InsertionSort()’ is the method implemented to sort the linked list with Insertion sort algorithm.
* ‘SelectionSort()’ is the method implemented to sort the linked list with Selection sort algorithm.
* To implement Merge sort algorithm for doubly linked list, the required methods are: Split(), GetTail(), mergeSort(), Merge() and MergeSort().
* To implement Quick sort algorithm for doubly linked list, the required methods are: partition(), quickSort() and QuickSort().

1. Bubble Sort

* The code for Bubble Sort:

Text

Description automatically generated

Figure 3. Implementation of Bubble sort

* The Bubble sort algorithm works by repeatedly swapping adjacent elements if they are in the wrong order. The algorithm starts from the beginning of the list and compares each pair of adjacent elements. If the first element of a pair is greater than the second element, they are swapped. The algorithm continues this process until it has made a full pass through the list without making any swaps.
* The best case time complexity of Bubble sort is O(n) if the list is already sorted. In this scenario, the algorithm will simply traverse the list without making any swaps. The worst case time complexity of Bubble sort is O(n^2) if the list is in reverse order. In this scenario, the algorithm will need to make n-1 swaps for each of the n elements in the list, resulting in a total of n(n-1)/2 swaps.

1. Insertion Sort

* The code for Insertion Sort:

Text

Description automatically generated

Figure 4. Implementation of Insertion sort

* The Insertion sort algorithm sorts the list in place, starting from the second element and comparing it with the previous elements. If the current element is smaller than the previous elements, it is swapped with the previous element until it is in its proper place. The algorithm continues this process for each subsequent element in the list until the entire list is sorted.
* The best case time complexity of Insertion sort is O(n) if the list is already sorted. In this scenario, the algorithm will simply traverse the list without making any swaps. The worst case time complexity of Insertion sort is O(n^2) if the list is in reverse order. In this scenario, the algorithm will need to make n-1 swaps for each of the n elements in the list, resulting in a total of n(n-1)/2 swaps.

1. Selection Sort

* The code for Selection Sort:

Text

Description automatically generated

Figure 5. Implementation of Selection sort

* The Selection sort algorithm works by repeatedly finding the minimum element from the unsorted part of the list and swapping it with the first unsorted element. The algorithm starts from the beginning of the list and finds the minimum element from the remaining unsorted part of the list. It then swaps the minimum element with the first unsorted element. The algorithm continues this process until the entire list is sorted.
* The best case and worst case time complexity of Selection sort is O(n^2), which makes it inefficient for larger lists. In both scenarios, the algorithm needs to make n-1 comparisons for each of the n elements in the list, resulting in a total of n(n-1)/2 comparisons.

1. Merge Sort

* Merge(): The purpose of the Merge function is to merge two sorted linked lists, left and right, into a single sorted linked list. The Merge function is a crucial part of the Merge Sort algorithm as it merges two sorted sublists into a single sorted list.

A screen shot of a computer program

Description automatically generated with medium confidence

Figure 6. Implementation of Merge function

* Split(): The Split function is responsible for dividing a linked list into two parts based on the specified stepSize. It traverses the list and updates the necessary pointers to split the list at the desired position. It returns the pointer to the start of the second part of the split or nullptr if the desired position is beyond the length of the list.

A screen shot of a computer program

Description automatically generated with low confidence

Figure 7. Implementation of Split function

* GetTail(): The GetTail function provides a way to find the last node of a doubly linked list or a sub-list efficiently. It traverses the list until it reaches the end and returns the pointer to the tail node.

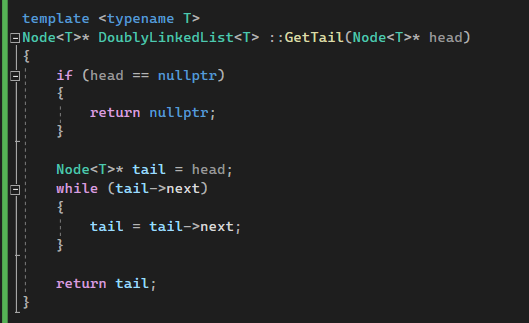


Figure 8. Implementation of GetTail function

* mergeSort(): The code implements an iterative merge sort algorithm for a doubly linked list. It splits the list into smaller subarrays, merges them in a sorted manner, and finally updates the head pointer to the sorted list. The reason i chose to implement merge sort using iterative approach instead of recursive approach is the iterative approach avoids stack overflow in some scenarios that the number of elements is large.

A picture containing text, screenshot, software, display

Description automatically generated

Figure 9. Implementation of mergeSort function

* MergeSort(): This function simply passes the head pointer of the doubly linked list into mergeSort() function to execute.

A picture containing text, font, screenshot, line

Description automatically generated

Figure 10. Implementation of MergeSort function

* Time Complexity: The merge sort algorithm has a time complexity of O(nlog n) in the average and worst cases, where n is the number of elements in the list. This is because the list is repeatedly divided into halves until individual elements are obtained, and then the sorted halves are merged back together.
* Space Complexity: The space complexity of the merge sort algorithm is O(1) because it only requires a constant amount of additional space for storing temporary pointers during the merging process. The algorithm performs sorting in-place without requiring additional memory proportional to the input size.

1. Quick Sort

* partition(): The purpose of the partition function is to rearrange the elements in a specific range of the doubly linked list such that all elements smaller than or equal to a chosen pivot value are placed to the left of the pivot, and all elements greater than the pivot are placed to the right of the pivot. The partition function plays a crucial role in the QuickSort algorithm by partitioning the list into smaller subarrays that can be independently sorted.

A picture containing text, screenshot, software

Description automatically generated

Figure 11. Implementation of partition function

* quickSort(): The function is the implementation of the QuickSort algorithm using an iterative approach with a stack. The quickSort function takes two parameters, head and tail, representing the range of nodes in the doubly linked list that need to be sorted. The algorithm uses a stack to keep track of subarrays that need to be sorted. It starts by pushing the initial range [head, tail] onto the stack. The algorithm enters a loop that continues until the stack is empty.

A picture containing text, screenshot, software

Description automatically generated

Figure 12. Implementation of quickSort function

* QuickSort(): This function simply passes the head pointer of the doubly linked list into quickSort() function to execute.

A picture containing text, font, screenshot

Description automatically generated

Figure 13. Implementation of QuickSort function

* Time Complexity: On average, QuickSort has a time complexity of O(n log n), where n is the number of elements in the list. However, in the worst-case scenario, when the pivot is consistently the smallest or largest element, the time complexity can degrade to O(n^2). This occurs when the list is already sorted or nearly sorted. Overall, QuickSort has good average-case performance and is widely used due to its efficiency.
* Space Complexity: The space complexity of this implementation is O(log n) due to the stack used to store subarray ranges. In the worst-case scenario, the stack can hold up to O(log n) elements when the partitioning is performed at each level of recursion. The actual space usage depends on the structure of the input list and the partitioning process.

1. Execution

* When the code is being executed, it first asks users to enter the number of elenments (size) of the list. Then, it asks users to type in the maximum number (range) of the list.
* For example:

A black background with white text

Description automatically generated with low confidence

In this example, the list has 100 elements and the range for each element is form 0 to 999. The system will randomly generate elements and store them in a vector.

* After that, the application prompts to ask in what condition you want to test the sorting algorithms. This is an example:

A black screen with white text

Description automatically generated with low confidence

* Finally, the application prints out the time it takes to sort the list with given size, range and condition. For example:

A screenshot of a computer program

Description automatically generated with medium confidence

1. **Comparison and Discussion**
2. The list is in random order

* We’ll take 4 cases to test the time taken by each

sorting method.

+ Case 1: size = 100, range = 1.000

A screenshot of a computer program

Description automatically generated with medium confidence

+ Case 2: size = 1.000, range = 10.000

A screenshot of a computer

Description automatically generated with medium confidence

+ Case 3: size = 10.000, range = 100.000

A screenshot of a computer

Description automatically generated with medium confidence

+ Case 4: size = 100.000, range = 1.000.000

A screenshot of a computer

Description automatically generated with medium confidence

* As can be easily seen in the cases above, Merge sort mostly takes the least time to sort the list (compared to 4 other sorting methods).
* Quick sort takes the highest time to sort the list when the size is small (100 elements). But as the size increases, quick sort shows that it is really effective since the time it takes to sort the list is so much less than bubble sort, insertion sort or selction sort.
* Following after quick sort is insertion sort. Insertion sort takes less time to sort than bubble sort and selection sort but still so much more than the orther 2 sorting algorithms.
* Bubble sort, in most scenario, takes the highest amount of time to sort the list.
* Selection sort takes less time to sort the list than Bubble sort but still more than Insertion sort.

1. The list is in ascending order (already sorted)

* We’ll take 4 cases to test the time taken by each

sorting method.

+ Case 1: size = 100, range = 1.000

A screenshot of a computer program

Description automatically generated with low confidence

+ Case 2: size = 1.000, range = 10.000

A screenshot of a computer program

Description automatically generated with medium confidence

+ Case 3: size = 10.000, range = 100.000

A screenshot of a computer program

Description automatically generated with medium confidence

+ Case 4: size = 100.000, range = 1.000.000

A screenshot of a computer

Description automatically generated with medium confidence

* In this scenario, bubble sort and insertion sort show a big difference, compare to orther 3 sorting algorithms. Bubble sort and insertion sort take so much less time to sort a sorted list and the reason for this is these 2 sorting techniques are adaptive. Which means with fully or partly sorted lists, they just simply traverse through all the sorted elements.
* The three orther sorting methods are not adaptive, therefore, they take much longer to sort. Merge sort still shows its effectiveness since the time it takes to sort the list is acceptable.

1. The list is in descending order (reversed order)

* We’ll take 4 cases to test the time taken by each

sorting method.

+ Case 1: size = 100, range = 1.000

A screenshot of a computer program

Description automatically generated with low confidence

+ Case 2: size = 1.000, range = 10.000

A screenshot of a computer

Description automatically generated with medium confidence

+ Case 3: size = 10.000, range = 100.000

A screenshot of a computer

Description automatically generated with medium confidence

+ Case 4: size = 100.000, range = 1.000.000

A screenshot of a computer

Description automatically generated with medium confidence

* This scenario is the worst case for most of the sorting algorithms. But for merge sort, since its time complexity for every case is the same (always O(nlogn)), it shows that it’s a very effective sorting algorithm for linked list in all case.
* The orther 4 algorithms are not really effective since in this scenario, they all have the time comlexities of O(n^2) and that makes them so much slower than merge sort.

1. **Conclusion**
2. Evaluation

* Among 5 given sorting methods, Merge sort is best for sorting a Doubly Linked List because of its time complexity and space complexity.
* The choice of sorting method for a doubly-linked list will depend on the specific requirements of the problem at hand. If the list is small or nearly sorted, Insertion sort may be a good choice due to its simple implementation and efficient performance for small lists. If the list is larger, a more efficient sorting method such as Quick sort or Merge sort may be more appropriate.

1. Conclusion

* This project provides a solid foundation for further exploration of sorting algorithms and their implementation in the C++ programming language. The implementation of the five sorting algorithms for doubly-linked lists highlights the differences in performance and efficiency of each algorithm, and provides a starting point for more advanced topics such as optimizing the algorithms for specific use cases. It also serves as an example of how to use templates and generic types in C++, which is a powerful tool for creating flexible and reusable code.
* In conclusion, the Insertion, Bubble, Selection, Merge and Quick sort algorithms provide a solid foundation for sorting doubly-linked lists in C++. While they may not be the most efficient algorithms for large lists, they are simple to understand and implement, making them a great starting point for learning about sorting algorithms.