COMP 2090SEF Project (Group 49)

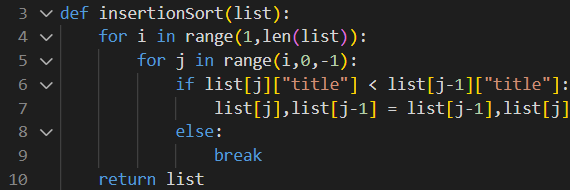
Study report (13490175 Chung Ming Lung)

**Introduction:**

For this project, I have selected “Tim sort” for the self-study sorting algorithm. My job in this group is to create the task page and it contains both sorting algorithms and storing data structure. Therefore, I will introduce the Tim sort algorithm as well as the data structure that I have implemented in the task page which is hash table.

**Sorting:**

Tim sort is a sorting algorithm that combines insertion sort and merge sort. The main idea of this sorting method is to utilize the advantage of insertion sort that has better performance when the data size is small. So, it also implements the technique of merge sort to separate a large data set into number of subsequences and sort them all with insertion sort and merge them back in one piece.



In the code I have first created an insertion sort for sorting sub-lists by comparing

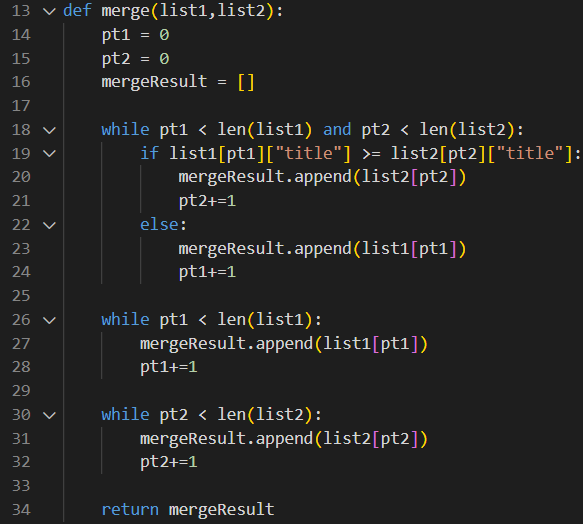
their titles since we are sorting the title of tasks. There will be 2 for loops the first one will

loop through the whole list except the element in the 0 index. So that there will not be an

error of comparing the first element with the last element. The second for loop is to put the

value of the i index in it’s place by comparing it with its previous index until the element

reaches a suitable position.



Then there is a merge function that will merge the 2 sorted list into one. First there

will be 2 pointers to check if all the elements of the list have been merged into a result list.

For example, [1,3,5,7,9] and [2,4,6]. When pointer 1 and 2 have not reach the length of

their allocated list they will do comparation between indexes of pointer 1 and 2.

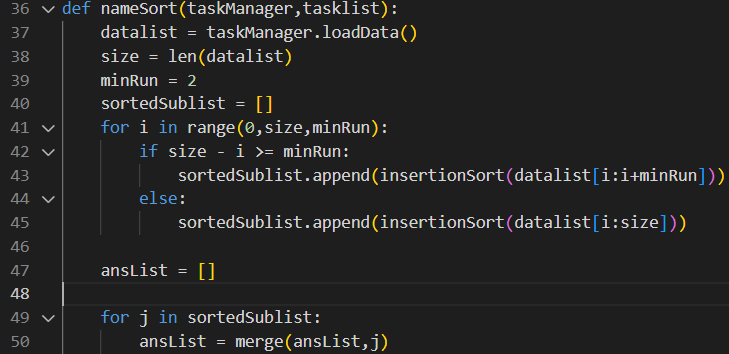
1<2 -> result = [1] every time the pointer with a smaller number will add 1, so that they can

use another number to compare instead of the merged one. Then 3>2 -> result = [1,2] and

pointer 2 will add 1. It will repeat until every element in both or one of the lists have

completely merged, then the remained list’s elements will be appended into the result list

directly.



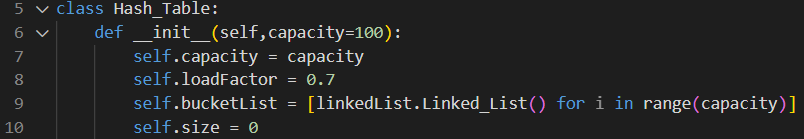
nameSort function is the main part of the sort which controls the whole algorithm by

calling the insertionSort and merge. First it will set a minimum run for the sorting algorithm which limits the size of the sub-list to 2. So, every sub-list will contain only 2 elements. Then, it uses the for loop to sort the sub-list by calling insertion sort in this case the time complexity of each insertion sort will be O(1). After that all sub-list will be merged into 1 list and be the result. Therefore, the best case will be O(n) if the list is sorted originally because there will not be extra steps for both merging and insertion sort and the for loop will break out instantly without swapping.

For the worst case. The part of sorting the sub-list needs O(1) \* O(n/2) and the merge part needs O(n log(n)) since we need n to iterate through all values while merging and log(n) time to merge 2 lists if there are 8 values in the list, the sub-list will be [[1,2],[4,5],[3,6],[7,8]] the first step will be [[1,2,4,5],[3,6],[7,8]], then [[1,2,3,4,5,6],[7,8]] finally [1,2,3,4,5,6,7,8] which is log 8 = 3 steps. The whole algorithm will be O(n log(n))+O(n/2) and O(n log(n)) is the most dominant part. Thus, it is the worst-case time complexity of the algorithm. In terms of the space complexity, there are 3 new list created to store values from the and 1 list for the original data. So, the space complexity is O(4n) and O(n) without the constant.

**Data Structure:**

**Hash Table:**

In this page, the hash table is the class with highest level to manage all data which is an Abstract Data Type.

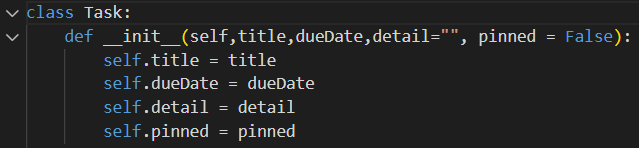
It contains attributes of capacity, load factor, bucket list, size (total record in the hash table) and methods including: loadData for getting all records attributes. \_resizeHashTable is for expanding the size of the bucket list to avoid collision to have a better searching performance. \_addSize add 1 to the size when adding new data. minusSize minus 1 from size when deleting data. \_checkLoadFactor check if the number of current existing records divided by the capacity has reached the load factor, and resize if it does. \_hashingFunction is for getting a suitable index for the data using sha256 hashing function from hashlib because the built-in hash() will return a different result every time. checkDuplicte is to check if there is a duplicate title in the record. hashData is for hashing the new task into the bucket. It will first use the hashing function to get the value then it will check the returned index of the hashing function to see if there is duplicate title using the check duplicate function, task will be added if no duplication is found. searchData is the searching function for user to search value by checking substring in the title, it might return more than one items. searchSingleData is different from searchData, it is for internal usage to search for the exact name of the task, so only 1 item will be returned. deleteTask is for deleting task. editTask and editPinned are for editing different attributes in the task object. Note that the searchData, searchSingleData and deleteTask function will implement the hash function to get the index directly instead of linear search. Therefore, the best case of searching will be O(1) while the worst case will still be O(n) if all the tasks are collided in the same bucket plus the linked list can only go by one in each iteration.

**Linked List:**

Each bucket in the hash table will be a linked list object to handle collisions. There are methods including append, delete, search, readSingleData, loadDate and checkDuplicate that are called by the hash table. There might be some functions that are the same with the hash table but different from the hash table those operations will be run inside of a bucket instead of the whole hash table. So the hash table can get the accurate item from the linked lists.The best and worst time complexity are still O(1) and O(n) depends on the location of the target record.

**Task Class:**

Each task is being created by a Task class and stored in a node of a lined list.



It contains title, due date for the task, detail of the task and pinned to see if it is important plus a pair of getters and setters.

**Conclusion:**

In conclusion, this is the time and space complexity of my self-study algorithm and the introduction of the hash table ADT with linked list and task object.

Self-Study Report (Heap Sort Algorithm)

Student Name: Lee Chung Hei

Student ID: 13496898

1. Introduction

In this self-study report, I have chosen the heap sort algorithm as the topic and used it to sort a list of unsorted dates in ascending order. This sorting technique uses the concept of a heap data structure and transforms the input list into a max heap. It then takes the largest element of the heap and swaps it to the end of the list. By repeating this process, the unsorted heap size will eventually decrease to one, and the list will be sorted. I will explain how I implemented this sorting algorithm in our project, its time and space complexity, and summarize the heap sort algorithm's overall performance.

2. Implementation

One of my tasks in the project is to sort a list of due dates set by users in the to-do list so that the tasks can show according to the ascending order of the dates. In this project, I decided to use the heap sort algorithm to sort a list called "dateDiff" that contains the corresponding day differences of the series of dates stored in another list. The reason that the "dateDiff" list is needed is that dates cannot be directly compared. Therefore, the datetime module has been used to find the differences in the number of days between the current date and each of the dates on the list (Figure 1).

The heap sort algorithm comprises two parts: heap construction and extraction.

Firstly, the "heapify()" function handles the construction of a max heap, which means swapping values of parent nodes with their child nodes if their left child or right child exists and is larger than the parent. This is done in a for loop so that all the parents of subtrees are swapped to be larger than or equal to their child to ensure that the root will eventually be the largest element in the list (Figure 2). The "dueDates" list that contains the actual dates will follow all the swapping processes of the "dateDiff" list that contains their day differences from the current date.

Secondly, the "heapSort()" function is used for calling the "heapify()" function to build the max heap (making the value of index 0 in the list the largest element). After that, the value of the root (index 0) will swap with the value of array index i. Making the largest element located at the end of the list. As the value at the root is no longer the maximum value, it calls the "heapify()" function again to build a new max heap. By repeating this process and reducing the "arrayLength" by one in each iteration, the whole array will be sorted (Figure 3).

An example of the running result of my implementation of the sorting algorithm is shown in Figures 4 and 5.

3. Complexity analysis

3.1 Time complexity

The heap sort algorithm has a consistent time complexity of O(N log N) in the best, average, and worst cases. This is because the heapify process ensures that the heap property of the structure is maintained and will carry out the same process regardless of the order of the elements at the beginning (Study Smarter, 2024). Due to its consistent time complexity, heap sort can perform ideally for large datasets (Alake, 2023).

3.2 Space complexity

The additional space required for heap sort has a complexity of O(1) if is implemented in the iterative approach, which means the space that the algorithm needs is constant. If it is implemented in the recursive approach, then it has the space complexity of O(log N), which will increase slowly when the input size increases (Shiksha Online, 2024). When comparing the space complexity of heap sort to other algorithms like merge sort which has a space complexity of O(N), it is relatively small. Furthermore, heap sort is an in-place sorting algorithm, which means it does not require additional space for temporary memory storage, which makes it more efficient regarding memory usage (Wikipedia contributors, 2024).

4. Conclusion

In conclusion, heap sort is an efficient algorithm when it is required to sort a large dataset because the time complexity of it remains constant and it also does not require extra memory space. However, it also has its disadvantages as it is relatively unstable, because if two elements are the same in the list, then their order may not be preserved when it is sorted the next time. Therefore, it depends on the requirements of the task to determine whether to use heap sort or other sorting algorithms.

5. Declaration

Heap sort is an algorithm that I have not learned before, so the knowledge about the concepts and the general implementation structure of the algorithm is from my self-studying from online notes and learning materials. I also asked AI questions about the algorithm for concepts that I did not understand to better understand the logic of the algorithm.

Reference

StudySmarter. (2024). *Heap Sort: Algorithm, Time Complexity.* StudySmarter UK

<https://www.studysmarter.co.uk/explanations/computer-science/algorithms-in-computer-science/heap-sort/>

Wikipedia contributors. (2024, April 15). Heapsort. In *Wikipedia, The Free Encyclopedia*.

Retrieved 10:13, April 15, 2024, from <https://en.wikipedia.org/w/index.php?title=Heapsort&oldid=1218978336>

Alake, R. (2023, April 4). *Heap Sort Explained.* Built In.

<https://builtin.com/data-science/heap-sort>

Shiksha Online. (2024, January 23). *Time and Space Complexity of Sorting Algorithms.*

Shiksha Online.

<https://www.shiksha.com/online-courses/articles/time-and-space-complexity-of-sorting-algorithms-blogId-152755>

‌ Appendix

A computer screen shot of a program code

Description automatically generated

(Figure 1)

Figure 1 is a code snippet of the creation of the "dateDiff" list using the imported datetime module. The "dateSort()" function calls the "heapsort()" function to sort the "dateDiff" list and the "dataList" will be sorted according to the "dateDiff" list.

A computer screen with text

Description automatically generated

(Figure 2)

A screen shot of a computer

Description automatically generated

(Figure 3)

The comments that I wrote in Figures 2 and 3 provide additional explanations for some lines of code.



A black background with white text

Description automatically generated

(Figure 4 & 5)

Figure 4 shows the original unsorted lists.

COMP 2090SEF Project (Group 49)

Study report: Shell sort

Name: LEE Shing Chun

Student ID: 13489717

Why choose Shell sort?

Shell Sort is an in-place comparison sort algorithm that improves the performance of insertion sort on larger lists. It starts by sorting elements that are far apart and gradually reduces the gap between elements to be compared. By using larger increments initially, Shell Sort can quickly move elements towards their correct positions, making the final insertion sort more efficient.

Explanation of the Code:

The Shell Sort algorithm begins by calculating the initial gap, which is typically set to half the length of the list. It then proceeds to repeatedly divide the gap by 2 until it reaches 1. Within each iteration of the gap, the algorithm performs an insertion sort on sublists, where elements are compared and swapped if necessary.

The provided code implements the Shell Sort algorithm within the shell sort method of a class. Let's break down the code step by step:

1, n = len(self.items): This line determines the length of the list self.items, which represents the collection of items to be sorted.

2, gap = n // 2: The initial gap is set to half the length of the list. This determines the distance between elements that will be compared and swapped during each iteration.

3, while gap > 0:: This loop continues until the gap is reduced to 0.

4, for i in range(gap, n):: This loop iterates over the list starting from the current gap position and up to the end of the list.

5, temp = self.items[i]: The current element at index i is temporarily stored in the temp variable.

6, j = i: A new index j is set to the value of i.

7, while j >= gap and self.items[j - gap].price > temp.price:: This loop compares the current element with the element gap positions before it. If the elements are out of order (based on the price attribute of the objects), they are swapped.

8, self.items[j] = self.items[j - gap]: If the above condition is true, the element at index j - gap is moved forward by gap positions.

9, j -= gap: The index j is decremented by the value of the gap.

10, self.items[j] = temp: The initial element (temp) is inserted at the correct position in the sublist.

11, gap //= 2: After each iteration, the gap is halved by integer division (//).

Conclusion:

Shell Sort is a smart way of sorting things. It starts by dividing the items into groups and sorts them separately. Then, it gradually brings the groups closer together and keeps sorting until everything is in the right order. This method helps sort things faster than just comparing and swapping items one by one.

Reference

<https://www.geeksforgeeks.org/shellsort/>

<https://www.javatpoint.com/shell-sort>

<https://hackmd.io/@Aquamay/H1nxBOLcO/https%3A%2F%2Fhackmd.io%2F%40Aquamay%2FrkgO8fpcu>

COMP 2090 individual report

Name : Cheng Long Hin

Student number: 13492410

**Introduction**

My chosen sorting algorithm is radix sort, which is a non-comparative integer sorting algorithm to sort the data. It groups the keys by individual digits which share the same position and value. The procedure of the sorting is from the least significant digit to the most significant digit. Radix sort uses counting sort as a subroutine.

**Time Complexity**

The time complexity of radix sort is O(d\*(n+b))

d is the number of digits

n is the number of elements

b is the base of numbering system used

1. 10 for decimal representation
2. 26 for alphabetic characters
3. 256 for ASCII characters

**The development of Radix sort**

Radix sort common use as sorting punch cards in 1900s. With the rapid development, the computing has been required to use this sorting algorithm, it can help us to sort the large amount of data. The example of the application is shopping list.

**The code related the Radix sort**

1. ‘radix\_sort()’ will identify the maximum length of the item names
2. ‘counting\_sort()’ will count the character position from the least significant to most significant

**The reason of choosing Radix sort**

1. It is efficient, radix sort will handle the number of item names including the length names and the number of items.
2. It is stable, radix sort will maintain the items at the same time with equal values.
3. It is usable, radix sort can improve the user experience like allowing quick location and selection of items.

**Conclusion**

Radix sort is suitable for the shopping list application based on its efficiency, stability and usability. We can save the time to sort the large number of the items, so that we hope that the sorting algorithm can help the user more convenience to use the shopping list application.

**References**

GfG. (2024, February 16). *Radix Sort Data Structures and Algorithms Tutorials*. GeeksforGeeks. <https://www.geeksforgeeks.org/radix-sort/>

S, R. A. (2023, February 14). *What Is Radix sort Algorithm: Pseudocode, Time Complexity, C & More*. Simplilearn.com. <https://www.simplilearn.com/tutorials/data-structure-tutorial/radix-sort>

D, N. (2019, August 31). *The first sorting algorithm and the founding of IBM. . .* <https://www.linkedin.com/pulse/first-sorting-algorithm-founding-ibm-nilesh-d>