MEARGEABLE HEAPS WITH LINKED-LISTS

Alexander Furman

June 2, 2024

Contents

1 Introduction **3**

2 Problem Statement **3**

3 Linked Lists **3**

3.1 Implementation . . . 4

4 Algorithm Design **4**

4.0.1 Minimum Invariant . . . 4

4.1 Heap with Sorted Input . . . 5

4.1.1 Order Invariant . . . 5

4.1.2 Insert . . . 5

4.1.3 Extract Minimum . . . 5

4.1.4 Minimum . . . 5

4.1.5 Union . . . 6

4.2 Heap with Unsorted Input . . . 6

4.3 Insert . . . 6

4.3.1 Extract Minimum . . . 7

4.3.2 Minimum . . . 7

4.3.3 Union . . . 7

4.4 Comparison . . . 8

4.5 Sorting . . . 8

4.5.1 Merge Sort . . . 8

5 Code and Examples **10**

5.1 Command-Line Interface . . . 10

5.2 Unsorted Heap . . . 11

5.2.1 Insert . . . 11

5.2.2 Minimum . . . 11

5.2.3 ExtractMin . . . 11

5.2.4 Union . . . 12

5.2.5 Sort . . . 12

5.3 Sorted Heap . . . 13

5.3.1 Insert . . . 13

5.3.2 Minimum . . . 13

5.3.3 ExtractMin . . . 13

5.3.4 Union . . . 14

6 Conclusion **14**

1 Introduction

Heaps are a fundamental data structure in computer science. They are commonly used for implementing priority queues[1], which serves many applications, including graph algorithms[2]. Traditionally, a heap is implemented with some tree-structure - utilising the hierarchical structure of which allows for efficient operations such as lookup, insertion, deletion.

Traditionally, a heap allows the user to perform the following operations:

- peakMin() / peakMax() – returning min/max element in the heap
- extractMin() / extractMax() – removing min/max element in the heap
- insert() – adds an element to the heap

Internal methods are used to maintain specific invariants about the structure of the tree, such as heapify() - a recursive method which ensures an arbitrary heap remains a max/min heap.

A mergeable heap is an extension of a vanilla heap, which also allows the user to perform union() - which merges one heap into another.

In this assignment, we explore a unique variation of the heap: a mergeable heap internally using a linked-list, instead of a tree. The challenge lies in the linearity of the linked-list data structure, and requires the design of different algorithms which can efficiently perform the aforementioned operations, with this new linear constraint.

2 Problem Statement

- We are tasked with implementing a mergeable heap which uses a linked-list as its basis for data storage.
- The user may specify whether their inputted values are sorted or unsorted.
- We assume the heap will hold integers

3 Linked Lists

Linked-Lists[1] are dynamic linear structures composed of nodes, each holding data and a reference to the next node. They are particularly useful in situations where many insertions/deletions are performed. However, it takes linear time when searching for an element in the list.

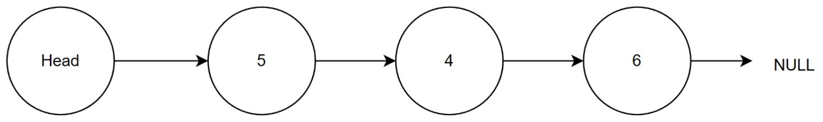

Figure 1: Example of Linked-List

3.1 Implementation

The specific implementation of linked-list used in this project is a doubly-linked-list, which allows the user to access and remove both the front and back nodes of the list in constant time. This is possible since the nodes have pointers to both next and previous elements in the list. Furthermore, a dummy-node is used for organization purposes. The data of the dummy element is set to ∞ for easy recognition.

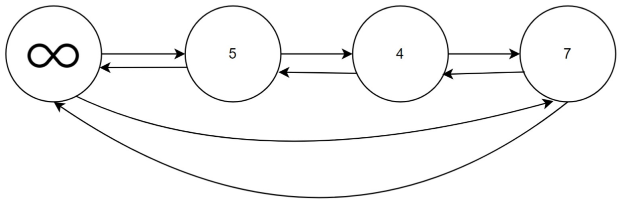

Figure 2: Implementation of Doubly-Linked-List Used in Project

4 Algorithm Design

The algorithms that are implemented can take advantage of known qualities of the input. Therefore, the algorithm design will be split between algorithms for sorted input, and algorithms for unsorted input.//

4.0.1 Minimum Invariant

When the user performs an operation on the heap, we require the minimum element to always be located in the same place within the doubly-linked-list. We arbitrarily decide that the minimum element should always be the first element

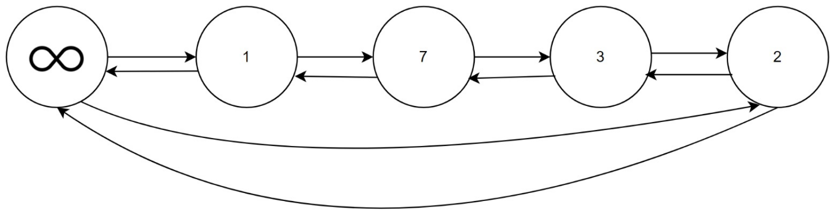

Figure 3: Example of a Linked-List Preserving the Minimum Invariant

4.1 Heap with Sorted Input

4.1.1 Order Invariant

Since the input is sorted, it would be beneficial to maintain a sorted doubly-linked-list. This will allow us to reduce the time complexity of certain algorithms.

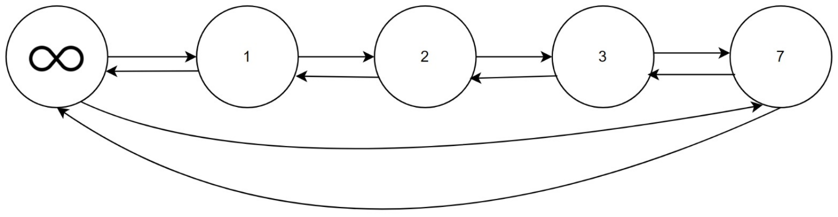

Figure 4: Example of a Linked-List Preserving the Order Invariant

4.1.2 Insert

Algorithm 1: Insertion Operation

```
1 Function Insert(SELF, value):
2     Append value to the end of SELF.list;
3     Return;
```

Since we know the input is monotonically increasing in value, we append each new value to the end of the list. This preserves the Order Invariant. The operation utilises the structure of the doubly-linked-list, and has a time complexity of $O(1)$.

4.1.3 Extract Minimum

Algorithm 2: Extraction Operation

```
1 Function ExtractMin(SELF):
2     Remove and return the first element of the SELF.list;
```

The operation makes use of the Minimum Invariant and removes the first element. After this removal, according to the Order Invariant, the new first element is still the minimum. The operation utilises the structure of the doubly-linked-list, and has a time complexity of $O(1)$.

4.1.4 Minimum

Algorithm 3: Minimum Operation

```
1 Function Minimum(SELF):
2     Return the first element of the SELF.list;
```

The operation makes use of the Minimum Invariant and peaks at the first element. The operation utilises the structure of the doubly-linked-list, and has a time complexity of $O(1)$.

4.1.5 Union

Algorithm 4: Union Operation

```
Function Union(SELF, heapB):
    Create a new helperHeap;
    $min \leftarrow$ SELF.ExtractMin();
    $minB \leftarrow$ heapB.ExtractMin();
    while $min \neq \infty$ or $minB \neq \infty$ do
        if $min < minB$ then
            healperHeap.Insert($minA$);
            $min \leftarrow$ SELF.ExtractMin();
        end
        else
            helperHeap.Insert($minB$);
            $minB \leftarrow$ heapB.ExtractMin();
        end
    end
    Assign SELF to be helperHeap;
```

The algorithm behaves as follows:

- Initializes a helper heap in constant time.
- Iteratively extracts and inserts elements from both SELF and heap B into the helper heap, with a time complexity of $O(n + m)$, where n is the number of elements in A and m is the number of elements in B.
- Assigns the helper heap to SELF, with a time complexity of either constant time for a shallow copy or $O(n + m)$ for a deep copy.
- Overall time complexity: $O(n + m)$.

The Minimum Invariant and Order Invariant are preserved, since we take care to insert values into the helper-heap in ascending order.

4.2 Heap with Unsorted Input

4.3 Insert

Algorithm 5: Insertion Operation

```
Function Insert(SELF, value):
    if SELF.list is empty or value < SELF.Minimum() then
        Insert value at the beginning of SELF.list
    end
    else
        Append value to the end of SELF.list
    end
```

This algorithm preserves the Minimum Invariant, since only if the inputted value is smaller than the current minimum, it is placed at the beginning of the list. The operation utilises the structure of the doubly-linked-list, and has a time complexity of $O(1)$.

4.3.1 Extract Minimum

Algorithm 6: Extraction Operation

```
Function ExtractMin(SELF):
    $min \leftarrow SELF.Minimum()$;
    Remove the first element of the SELF.list;
    Create healperHeap;
    while SELF is not empty do
        $val \leftarrow$ extract first element from SELF.list;
        helperHeap.Insert(val);
    end
    Assign SELF to be helperHeap;
    return min
```

This algorithm behaves as follows:

- call SELF.Minimum() and assign it to min. This takes $O(1)$ time.
- Remove first item in SELF.list. this takes $O(1)$ time.
- Create a new helperHeap. this takes $O(1)$ time.
- iteratively extract and insert all values from SELF to helperHeap. This takes this takes $O(n)$ time.
- Assign SELF to be helperHeap. this takes $O(1)$ time for a shallow-copy, or $O(n)$ time for a deep-copy.
- return min - $O(1)$.
- Overall time complexity: $O(n)$

After extracting the minimum value in the heap, the next value is not necessarily the minimum. Therefore, we remove all values from SELF, and use Insert() to add them to helperHeap - and finally set SELF to helperHeap. This ensures that the new minimum is located at the front of SELF.list, thereby preserving the Minimum Invariant.

4.3.2 Minimum

This operation is exactly the same as the heap with sorted input. This is because both the sorted and unsorted version of the heaps maintain the same Minimum Invariant - that the minimum value is located at the front of SELF.list.

4.3.3 Union

Algorithm 7: Union Operation

```
Function Union(SELF, heapB):
    while heapB is not empty do
        $val \leftarrow$ extract value at front of heapB.list;
        SELF.Insert(val);
    end
```

This algorithm iteratively removes the first value in heapB.list, and uses Insert() to add it to SELF - overall taking $O(m)$ time complexity, where m is the number of elements in heapB. This preserves the Minimum Invariant in SELF. Note that during this operation, heapB does not maintain the Minimum Invariant, however by the end of the operation, it is empty - once more preserving the invariant. Note that after this union, the ExtractMin operation will now be $O(n + m)$.

4.4 Comparison

Here are the time complexities of each operation described above:

Table 1: Algorithm Comparison

Algorithm	Input Type	Time Complexity
Insert()	Ordered	$O(1)$
ExtractMin()	Ordered	$O(1)$
Minimum()	Ordered	$O(1)$
Union()	Ordered	$O(n + m)$
Insert()	Unordered	$O(1)$
ExtractMin()	Unordered	$O(n)$
Minimum()	Unordered	$O(1)$
Union()	Unordered	$O(m)$

We see that Insert and Minimum operations are the same complexity for both types of inputs. However we can perform ExtractMin faster when using sorted input. Finally, the Union operation is slightly faster for unsorted input - however for values of m comparable to n, this is negligible. Furthermore, the ExtractMin operation for the unsorted heap grows to $O(n + m)$ whereas the sorted heap's operation remains constant.

Therefore, we see that it is preferred to have sorted input to the heap.

4.5 Sorting

Given the importance of sorted input, it's advantageous to develop an algorithm to sort an unsorted heap. By sorting the heap, we transform its list into a monotonically increasing sequence of elements. Once the heap is sorted, we can leverage the benefits of having organized data, and speed up certain operations.

While we do want to sort the heap, we want to find a sorting method for a linked list which has a relatively good time complexity.

4.5.1 Merge Sort

The Merge Sort algorithm is a divide and conquer algorithm traditionally used for sorting arrays of values. It has a time complexity of $O(nlgn)$ in both best and worst cases, and it is also a stable sorting algorithm.

We will see that a time complexity of $O(nlgn)$ with a linked-list is still achievable, with a few minor updates to the algorithm.

```
Algorithm 8: Merge Operation
Function Merge(list1, list2, mergedList):
    $minimum1 \leftarrow$ extractAtIndex(list1, 0);
    $minimum2 \leftarrow$ extractAtIndex(list2, 0);
    while $minimum1 \neq INF$ or $minimum2 \neq INF$ do
        if $minimum2 = INF$ or $minimum1 < minimum2$ then
            append(mergedList, $minimum1$);
            $minimum1 \leftarrow$ extractAtIndex(list1, 0);
        end
        else
            append(mergedList, $minimum2$);
            $minimum2 \leftarrow$ extractAtIndex(list2, 0);
        end
    end
```

The Merge operation is a helper-function for Merge Sort. It combines two sorted doubly-linked-lists into a single doubly-linked-list. It extracts the first element from list1 and list2. Then it iteratively compares the minimums: If minimum1 ¡ minimum then add minimum1 to mergedList, and extract the next minimum value from list1 and append it to minimum1. Otherwise, do the same for minimum2 and list2. The time complexity for this algorithm is $O(n + m)$, since we extract and insert each value in list1 and list2 exactly once.

```
Algorithm 9: Merge Sort Algorithm
Function MergeSort(inputList):
    $inputListSize \leftarrow$ inputList.getSize();
    if $inputListSize = 1$ then
        return;
    end
    $subList\_L \leftarrow$ create empty DoublyLinkedList;
    $subList\_R \leftarrow$ create empty DoublyLinkedList;
    $middleIndex \leftarrow inputListSize \div 2$;
    $count \leftarrow 0$;
    while $count < middleIndex$ do
        $subList\_L$.append(inputList.extractAtIndex(0));
        $count \leftarrow count + 1$;
    end
    while $inputList.getSize() > 0$ do
        $subList\_R$.append(inputList.extractAtIndex(0));
    end
    MergeSort(subList_L);
    MergeSort(subList_R);
    Merge(subList_L, subList_R, inputList);
```

This version of the Merge Sort algorithm is almost identical to the original from the textbook[1], with a few small changes:

- The function MergeSort takes an unsorted doubly-linked-list inputList as input.
- It checks if inputList has only one element, in which case it's already sorted, so the function returns - $O(1)$.
- Otherwise:
  - It extracts the first half of elements from inputList, and appends them to subList_L.
  - It extracts the remaining half of the elements of inputList, and appends them to subList_R.
  - It recursively calls MergeSort() on both subList_L and subList_R.
  - Finally, it merges the sorted subList_L and subList_R back into inputList.

The recursive relation of this algorithm can be expressed as $T[n] = 2(T[n/2]) + O(n)$.

- T[n]: The complexity of MergeSort applied to doubly-linked-list of size n
- 2(T[n/2]): The complexity of sorting two sublists of size n/2.
- : O(n): The complexity of transferring all elements from list of size n to two sublists of size n/2, plus the complexity of transferring all elements from the sublists back to the main list during Merge ($O(n + n) = O(n)$)

According to the Master Method, the total complexity of this algorithm is $O(nlgn)$[1].

5 Code and Examples

In this report's accompanying code, the user is able to experiment with implementations of the mergeable heaps described above (both for sorted and unsorted user input).

5.1 Command-Line Interface

The user can execute the following commands in the terminal:

```
make heap {SORT_TYPE} {NAME}      // Creates a new heap (either sorted or unsorted)
heap {NAME} load {FILENAME}       // Inserts values from a file into the heap
heap {NAME} insert {INT}          // Inserts an integer into the selected heap
heap {NAME} minimum               // Prints the minimum value in the heap
heap {NAME} extractmin            // Removes the minimum value in the heap and prints it
heap {NAME} sort                  // Sorts the current contents of the heap
heap {NAME_1} union heap {NAME_2} // Merges heap_2 into heap_1
heap {NAME} display               // Displays the heap
help                              // Displays menu of commands
quit                              // Quits the program
```

5.2 Unsorted Heap

5.2.1 Insert

Here's an example of inserting the list [9 1 2 7] into an unordered heap:

```
Enter a command ('quit' to exit // 'help' for menu): make heap unsorted A
Heap A (unsorted input) created!
Enter a command ('quit' to exit // 'help' for menu): heap A load testing/2.txt
Inserted 9 into heap A (unsorted input)!
Displaying heap A (unsorted input): HEAD -> 9
Inserted 1 into heap A (unsorted input)!
Displaying heap A (unsorted input): HEAD -> 1 -> 9
Inserted 2 into heap A (unsorted input)!
Displaying heap A (unsorted input): HEAD -> 1 -> 9 -> 2
Inserted 7 into heap A (unsorted input)!
Displaying heap A (unsorted input): HEAD -> 1 -> 9 -> 2 -> 7
```

Notice that only when the inputted value was smaller than the first element, it was placed in the front. Otherwise it was placed in the back.

5.2.2 Minimum

Here's an example of finding the minimum in an unsorted heap

```
Displaying heap A (unsorted input): HEAD -> 1 -> 9 -> 2 -> 7
Enter a command ('quit' to exit // 'help' for menu): heap A minimum
Minimum of heap A (unsorted input) = 1
Displaying heap A (unsorted input): HEAD -> 1 -> 9 -> 2 -> 7
Enter a command ('quit' to exit // 'help' for menu):
```

Simply returns the first element in the list

5.2.3 ExtractMin

Here's an example of performing ExtractMin on an unordered heap:

```
Enter a command ('quit' to exit // 'help' for menu): heap A display
Displaying heap A (unsorted input): HEAD -> 1 -> 9 -> 2 -> 7
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (unsorted input): 1
Displaying heap A (unsorted input): HEAD -> 2 -> 9 -> 7
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (unsorted input): 2
Displaying heap A (unsorted input): HEAD -> 7 -> 9
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (unsorted input): 7
Displaying heap A (unsorted input): HEAD -> 9
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (unsorted input): 9
Displaying heap A (unsorted input): HEAD
```

Notice how after each call of ExtractMin, the remaining heap is reordered in order to keep the minimum value at the front of the doubly-linked-list

5.2.4 Union

Here we see an example of the Union operation between two unsorted heaps

```
Enter a command ('quit' to exit // 'help' for menu): heap A display
Displaying heap A (unsorted input): HEAD -> 2 -> 5 -> 3 -> 8
Enter a command ('quit' to exit // 'help' for menu): heap B display
Displaying heap B (unsorted input): HEAD -> 1 -> 9 -> 2 -> 7
Enter a command ('quit' to exit // 'help' for menu): heap A union heap B
Merged heap B (unsorted input) into heap A (unsorted input)
Displaying heap A (unsorted input): HEAD -> 1 -> 2 -> 5 -> 3 -> 8 -> 9 -> 2 -> 7
Displaying heap B (unsorted input): HEAD
```

Notice that only the first element of B was placed at the beginning of A's list, whereas the rest were placed at the end.

5.2.5 Sort

Here we see an example of the Sort operation applied on an unsorted heap

```
Displaying heap A (unsorted input):
HEAD -> 2 -> 5 -> 15 -> 21 -> 27 -> 9 -> 20 -> 45 -> 28 -> 14 -> 47 -> 16 -> 30 -> 25 -> 44
Enter a command ('quit' to exit // 'help' for menu): heap A sort
Here is the sorted contents of heap A (unsorted input):
HEAD -> 2 -> 5 -> 9 -> 14 -> 15 -> 16 -> 20 -> 21 -> 25 -> 27 -> 28 -> 30 -> 44 -> 45 -> 47
Enter a command ('quit' to exit // 'help' for menu):
```

We see that Merge Sort successfully sorted the unsorted heap.

5.3 Sorted Heap

5.3.1 Insert

Here's an example of inserting the list [1 2 2 6 49 100] into a sorted heap:

```
Enter a command ('quit' to exit // 'help' for menu): make heap sorted A
Heap A (sorted input) created!
Enter a command ('quit' to exit // 'help' for menu): heap A load testing/8.txt
Inserted 1 into heap A (sorted input)!
Displaying heap A (sorted input): HEAD -> 1
Inserted 2 into heap A (sorted input)!
Displaying heap A (sorted input): HEAD -> 1 -> 2
Inserted 2 into heap A (sorted input)!
Displaying heap A (sorted input): HEAD -> 1 -> 2 -> 2
Inserted 6 into heap A (sorted input)!
Displaying heap A (sorted input): HEAD -> 1 -> 2 -> 2 -> 6
Inserted 49 into heap A (sorted input)!
Displaying heap A (sorted input): HEAD -> 1 -> 2 -> 2 -> 6 -> 49
Inserted 100 into heap A (sorted input)!
Displaying heap A (sorted input): HEAD -> 1 -> 2 -> 2 -> 6 -> 49 -> 100
```

In this implementation, the elements are automatically placed at the end of the doubly-linked-list

5.3.2 Minimum

Here's an example of finding the minimum in a sorted heap

```
Displaying heap A (sorted input): HEAD -> 1 -> 2 -> 2 -> 6 -> 49 -> 100
Enter a command ('quit' to exit // 'help' for menu): heap A minimum
Minimum of heap A (unsorted input) = 1
Displaying heap A (unsorted input): HEAD -> 1 -> 2 -> 2 -> 6 -> 49 -> 100
```

Simply returns the first element in the list

5.3.3 ExtractMin

Here's an example of performing ExtractMin on a sorted heap:

```
Displaying heap A (sorted input): HEAD -> 1 -> 2 -> 2 -> 6 -> 49 -> 100
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (sorted input): 1
Displaying heap A (sorted input): HEAD -> 2 -> 2 -> 6 -> 49 -> 100
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (sorted input): 2
Displaying heap A (sorted input): HEAD -> 2 -> 6 -> 49 -> 100
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (sorted input): 2
Displaying heap A (sorted input): HEAD -> 6 -> 49 -> 100
```

```
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (sorted input): 6
Displaying heap A (sorted input): HEAD -> 49 -> 100
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (sorted input): 49
Displaying heap A (sorted input): HEAD -> 100
Enter a command ('quit' to exit // 'help' for menu): heap A extractmin
Extracted minimum from heap A (sorted input): 100
Displaying heap A (sorted input): HEAD
```

In this implementation, the first element is removed, and the new minimum is always the first element automatically.

5.3.4 Union

Here's an example of performing Union on two sorted heaps:

```
Enter a command ('quit' to exit // 'help' for menu): heap A display
Displaying heap A (sorted input): HEAD -> 1 -> 2 -> 2 -> 6 -> 49 -> 100
Enter a command ('quit' to exit // 'help' for menu): heap B display
Displaying heap B (sorted input): HEAD -> 0 -> 3 -> 7 -> 200
Enter a command ('quit' to exit // 'help' for menu): heap A union heap B
Merged heap B (sorted input) into heap A (sorted input)
Displaying heap A (sorted input): HEAD -> 0 -> 1 -> 2 -> 2 -> 3 -> 6 -> 7 -> 49 -> 100 -> 200
Displaying heap B (sorted input): HEAD
Enter a command ('quit' to exit // 'help' for menu):
```

Here we see that after the Union operation, the merged heap is still sorted.

6 Conclusion

In conclusion, we successfully implemented mergeable heaps using a doubly-linked-list as the underlying data structure. We explored two different implementations, one for sorted input and another for unsorted input. We developed algorithms for the Insert, ExtractMin, Minimum, and Union operations for both implementations. We also implemented a version of the Merge-Sort algorithm for sorting unsorted heaps. We analyzed the time complexities of these algorithms and provided examples demonstrating their functionality.

References

[1] R. R. C. S. Thomas H. Cormen, Charles E. Leiserson, *Introduction to Algorithms.* MIT Press, 1989.

[2] O. Salzman and D. Halperin, “Asymptotically near-optimal rrt for fast, high-quality, motion planning,” 2013.