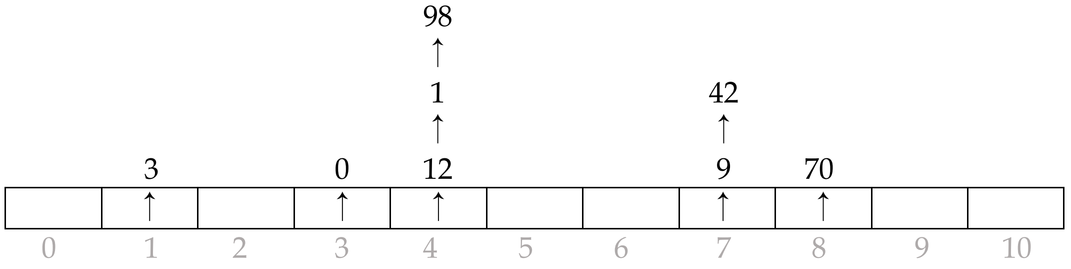
Cpt_s 233 HW3

Boxiang Lin

1. HashTable: haskey(key) = (key*key + 3) % 11
   - Separate chain



   - Linear probing: offset = probe(i) = (i+1) % tablesize, haskey+offset

	3		0	12	1	98	9	42	70	
0	1	2	3	4	5	6	7	8	9	10

   - Quadratic probing: offset = proble(i) = (i*i +5)% tablesize, hashkey+offset

	42		0	12		3	9	70	1	98
0	1	2	3	4	5	6	7	8	9	10

2. If we are using the technique of mod(%) with Table Size to calculate the haskKey, **using 101, the largest prime number among theses sizes, to be the table size will greatly prevent the clusters**.

   (by imaging each haskey in sequential $1, \ldots\ldots n_i$, n_{i+1}, each hashkey n_i and n_{i+1} differences is f): The characteristic of the prime number that has the only two factors 1 and itself, then the set of hasKey has a large range of probablities to get uniformly distributed by maintain ($|n_{i+1} - n_i| \neq f$, f $\in$ 101's factors) . While 500 has the factors of 1, 2, 4, 5, 10, 20, 25, 50, 100, 125, 250, 500. By a roughly estimate, if let f $\in$ 500's factors, then for set to maintain $|n_{i+1} - n_i| \neq f$ probabilities are estimated lower than let the size to be 101. Hence a reason primes so magical.

3. ..
   - Lambda = 53491/106963 = 0.5000888158
   - Yes, for linear probing technique we should rehash, since lambda is greater than 0.5.
   - No, for separate chaining technique, lambda not greater than 1.

4. Hash Table Big O

Function	Worst Complexity	Average Complexity
Insert(x)	O(N)	Θ(1)
Rehash()	O(N)	Θ(N)
Remove(x)	O(N)	Θ(1)
Contains(x)	O(N)	Θ(1)

5. **Didn’t see question 5 in PDF.**

6. **HashFunction**

```
int hasit ( int key, int TS){
        return key%TS;
}

int hasit ( String key, int TS){
        int keyVal = 0;
        char ch[] = key.toCharArray();
        for(int i=0; i<ch.length; i++){
                keyVal = keyVal+ch[i];  //get the sum of each character’s ascii code
        }
        return keyVal%TS;
}
```

7. **Problem in rehas();**
I think the problem is at the second nested for loop where we addElement from the oldArray, but the elements are still added to our new Array with the same HashKey from the oldArray instead of calculated a new HashKey by the hash function. I think this is the reason that possibly causes clusters when element numbers(entries) are large. Overall, this rehash method only resize but didn’t rehash the hashKey of the elements.

8. **Heap big O**

Function	Worst Big O
Push(x)	O(log N)
Top()	O(1)
Pop()	O(log N)
Buildheap	O(N log N) //Average: Θ (N)

9. **A good application for the Priority Queue may be:**
An e-commerce website like Amazon that has millions of seller, buyer, and products wants to provide great customer experiences. So, they have an evaluation department that investigating sellers that possibly involve in fraud or other illegal activities. They plan to find out the sellers that received the most report from buyers and begin investigating from there. So, they stored all seller’s information into a max binary heap structure based on the numbers of the bad report the sellers had and will investigating them level by level from top to bottom.

Since buildHeap average complexity is Θ(N), it is very efficient to build up the data set in Max binary heap, hence a advantage compare to other sorting algorithm.

10. **Parent: i/2**
Children: 2*i, 2*i + 1

D heap:
Parent: (i-2)/d+1

Children: (i-1)*d+j+1, where j is the nth of child

11. Inserting elements into a Min-Heap.

Insert 10

	10								

Insert 12

	10	12							

Insert 1 and percolate up

	1	12	10						

Insert 14

	1	12	10	14					

Insert 6 and percolate up

	1	6	10	14	12				

Insert 5 and percolate up

	1	6	5	14	12	10			

Insert 15

	1	6	5	14	12	10	15		

Insert 3 and percolate up

	1	3	5	6	12	10	15	14	

Insert 11

	1	3	5	6	12	10	15	14	11

12. BuildHeap

	1	3	5	11	6	10	15	14	12

13. deleteMin for the heap in Q.12

1st deleteMin

	3	6	5	11	12	10	15	14	

2nd deleteMin

	5	6	10	11	12	14	15		

3rd deleMin

	6	11	10	15	12	14			

14. Sorting Big O/Stability

Algorithm	**Average Complexity**	**Stable**
Bubble Sort	Θ(N^2)	Yes

Insertion Sort	Θ(N^2)	Yes
Heap Sort	Θ(N logN)	No
Merge Sort	Θ(N logN)	Yes
Radix Sort	Θ(kN)	Yes
Quick Sort	Θ(N logN)	No

15. Differences between Merge Sort and Quick Sort.

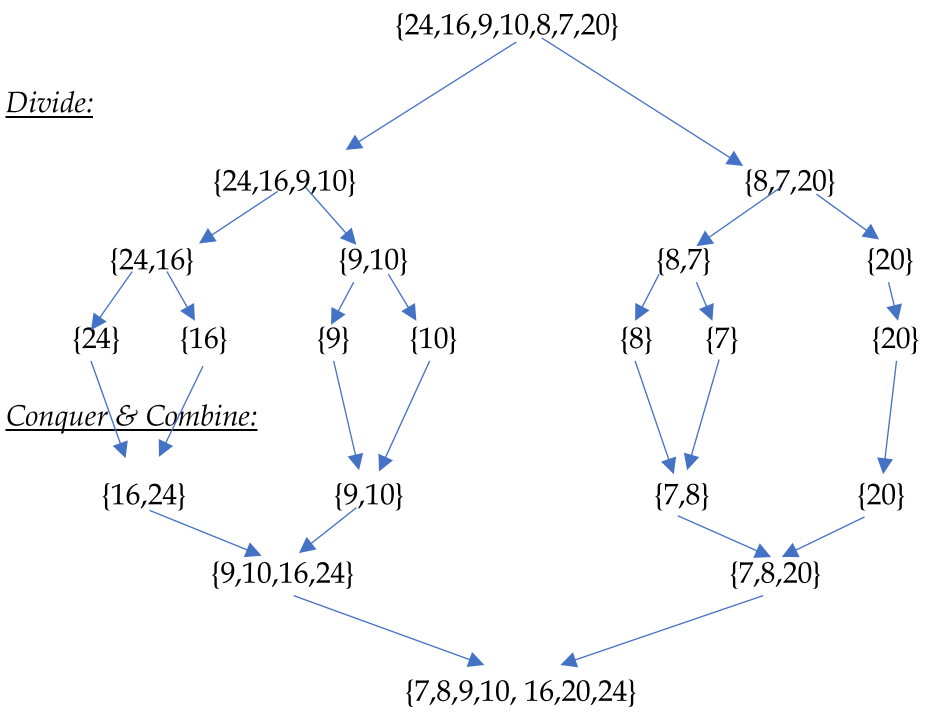
Key Diffrence:

- Merge Sort is stable and Quick Sort not stable.
- Quick sort using a pivot helper for divide and conquer technique, Merge sort just divide to half and half until one and then conquer up.
- Extra space O(N) need for Merge sort compared to the quick sort.
- Element comparisons of Quicksort > Element comparisons of Merge sort.
- From the experiment someone made from the online, it appears that quick sort is faster than merge sort.

Language chosen:

- JAVA: good to use Merge Sort since Java handling the memory.
- C++: good to use Quick Sort, C++ running speed faster.

16. Merge Sort



17. Quick sort "Green is the pivot"

24	16	9	10	8	7	20

24	16	9	20	8	7	10
L					R	

7	16	9	20	8	24	10
	L				R	

7	16	9	20	8	24	10
	L			R		

7	8	9	20	16	24	10
		L		R		

L>pivot stop, R--;

7	8	9	20	16	24	10
			L	R		

7	8	9	20	16	24	10
			L R			

7	8	9	20	16	24	10
		R	L			

When L>R, swap pivot to position L, now left elements of pivot smaller than pivot and right elements of pivot larger than pivot.

7	8	9	10	16	24	20

Quicksort to the left sub-list.

7	8	9	10	16	24	20

7	9	8	10	16	24	20
L	R					

Now L>pivot stop L, and R—

7	9	8	10	16	24	20
	L R					

7	9	8	10	16	24	20
R	L					

R>L, swap pivot to L position

7	8	9	10	16	24	20

Left and Right Sub-List of this left sub-list now only contains {7} and {9} we are done with it, then went to the previous right sub-list.

7	8	9	10	16	24	20

7	8	9	10	16	20	24
				L	R	

7	8	9	10	16	20	24
					L R	

7	8	9	10	16	20	24
					R	L

L>R, pivot and L position is same so pivot stay at L position, now consider its left sub-list.

7	8	9	10	16	20	24

7	8	9	10	20	16	24
				L R		

L stop, R--;

7	8	9	10	20	16	24
				L		

L>R at this time, swap pivot to the L position

7	8	9	10	16	20	24
7	8	9	10	16	20	24

Left only {20} sub list, so it is sorted.

7	8	9	10	16	20	24

Completed the quick sort.