<https://github.com/TT00FE39-3001/lecture4>

<https://github.com/seppotk/Datastructures_and_algorithms.git>

**Outline**

**Announcements**

* 2023-02-20: [Review: Optional Session](https://github.com/TT00FE39-3001/lecture-2023-02-20-review)
* 2023-02-27: Lecture 5
* 2023-03-10: Career coaching (OMA)
  + Consider: peer evaluation in social media
  + Remember: Degree first

**Topics**

* [Review](https://github.com/TT00FE39-3001/lecture4/blob/main/review.md)
* Divide & Conquer
  + Merge Sort
  + Quick sort
* Hash Tables
  + Space vs Time
  + Collision resolution
* Linked lists

**This Week in Points**

* Group Activities (Max 9 points)
* Homework (Max 9 points)
* Peer reviews (Max 7 points)

**Part 1: Divide & Conquer**

* [Merge Sort](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/Mergesort.html)
* [Quick sort](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/Quicksort.html)
* [Activity 1](https://github.com/TT00FE39-3001/lecture4/blob/main/activity1)

**Part 2: Hash Tables**

* Space vs Time
* Hash Tables
* Collision resolution
  + [Probability of a collision](https://opendsa-server.cs.vt.edu/ODSA/StandaloneModules/20200825204059/html/HashFunc.html)
  + [Open Hashing](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/OpenHash.html)
  + [Bucket Hashing](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/BucketHash.html)
* [Activity 2](https://github.com/TT00FE39-3001/lecture4/blob/main/activity2)

**Part 3: Linked Lists & Asymptotic Analysis**

* [Linked Lists](https://www.softwaretestinghelp.com/linked-list/)
* [Asymptotic Analysis and Upper Bounds](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/AnalAsymptotic.html)
* [Activity 3](https://github.com/TT00FE39-3001/lecture4/blob/main/activity3)

**# Outline**

**## Announcements**

- 2023-02-20: [Review: Optional Session](<https://github.com/TT00FE39-3001/lecture-2023-02-20-review>)

- 2023-02-27: Lecture 5

- 2023-03-10: Career coaching (OMA)

  - Consider: peer evaluation in social media

  - Remember: Degree first

**## Topics**

- [Review](./review.md)

- Divide & Conquer

  - Merge Sort

  - Quick sort

- Hash Tables

  - Space vs Time

  - Collision resolution

- Linked lists

**## This Week in Points**

- Group Activities (Max 9 points)

- Homework (Max 9 points)

- Peer reviews (Max 7 points)

**## Part 1: Divide & Conquer**

- [Merge Sort](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/Mergesort.html)

- [Quick sort](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/Quicksort.html)

- [Activity 1](./activity1)

**## Part 2: Hash Tables**

- Space vs Time

- Hash Tables

- Collision resolution

  - [Probability of a collision](https://opendsa-server.cs.vt.edu/ODSA/StandaloneModules/20200825204059/html/HashFunc.html)

  - [Open Hashing](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/OpenHash.html)

  - [Bucket Hashing](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/BucketHash.html)

- [Activity 2](./activity2)

**## Part 3: Linked Lists & Asymptotic Analysis**

- [Linked Lists](https://www.softwaretestinghelp.com/linked-list/)

- [Asymptotic Analysis and Upper Bounds](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/AnalAsymptotic.html)

- [Activity 3](./activity3)

review

**# Review**

**## Classification of Algorithm techniques**

- Brute Force

  - Linear search

  - Bubble sort

  - Selection sort

- Decrease and conquer

  - Binary Search

  - Insertion sort

- Divide-and-Conquer

  - `pow()`

**## Data Structures & ADT**

- Arrays, Linked Lists

- Queues

- Stacks

**## Analysis of Algorithm Efficiency**

- Big O Complexity

- Average case vs worst case

- Space vs Time

**## Misc**

- FIFO vs LIFO

- Recursion vs Iteration

- Logarithms vs Exponential

links

**# Links**

- [QuickSort](https://www.geeksforgeeks.org/quick-sort/)

- [Merge Sort Algorithm](https://www.geeksforgeeks.org/merge-sort/)

- [Data Structures and Algorithms ](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/)

- [Hashing](https://www.geeksforgeeks.org/hashing-data-structure/)

- https://cpp.sh/

HOMEWORK

**# Homework**

**## Task 1/3:Videos**

- [Hashes 1](https://youtu.be/)

- [Hashes 2](https://youtube.com/watch?v=XYmI-T-JJso&si=EnSIkaIECMiOmarE)

- [Hashes 3](https://youtube.com/watch?v=YIoZQwWJIDA&si=EnSIkaIECMiOmarE)

- [Hashes 4](https://youtube.com/watch?v=jtMwp0FqEcg&si=EnSIkaIECMiOmarE)

**## Task 2/3: Reading**

- [Quick Sort In C++](https://www.softwaretestinghelp.com/quick-sort/)

- [Merge Sort In C++](https://www.softwaretestinghelp.com/merge-sort/)

- [Hash Table In C++](https://www.softwaretestinghelp.com/hash-table-cpp-programs/)

- [Linked List Operations](https://www.softwaretestinghelp.com/linked-list/)

- [Asymptotic Analysis and Upper Bounds](https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/AnalAsymptotic.html)

**## Task 3/3: Pre-Lecture (Videos)**

- [Trees and heaps](https://youtube.com/watch?v=lhTCSGRAlXI&si=EnSIkaIECMiOmarE)

- [Heaps 1](<https://youtube.com/watch?v=BzQGPA_v-vc&si=EnSIkaIECMiOmarE>)

ACTIVITY1

**# Activities**

**## Task 1**

- Refer to the following link. Discuss how Merge-sort works:

<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Sorting/mergesortAV.html>

- Merge-sort Practice. Refer to the following link. Merge the two sub-arrays into the larger array.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Sorting/MergesortMergePRO.html>

**## Task 2**

- Refer to the following link. Discuss how Quick-sort works:

<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Sorting/quicksortAV.html>

- Quick-sort Practice. Refer to the following link. Partition the array using quicksort.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Sorting/QuicksortPartitPRO.html>

**## Task 3**

- The following snippet is from `./src/quicksort.cpp` lines 32-43. Discuss in groups how the code works:

```cpp

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        //partition the array

        int pivot = partition(arr, low, high);

        //sort the sub arrays independently

        quickSort(arr, low, pivot - 1);

        quickSort(arr, pivot + 1, high);

    }

}

```

- The following snippet is from `./src/quicksort.cpp` line 27. Discuss in groups how ìt works:

```cpp

swap(&arr[i + 1], &arr[high]);

```

**## Task 4: Individual (at home)**

1. Merge-sort has better worst case performance than quicksort. So why Quick-sort is considered better than Merge-sort? Refer to the following article

   https://www.geeksforgeeks.org/quicksort-better-mergesort/

2. Refer to the following article. Analyze the complexity of the Merge-sort algorithm.

   https://www.softwaretestinghelp.com/merge-sort/

3. Refer to the following article. Analyze the complexity of the Quick-sort algorithm.

   https://www.softwaretestinghelp.com/quick-sort/

ANSWERS:

**## Task 1**

- Refer to the following link. Discuss how Merge-sort works:

<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Sorting/mergesortAV.html>

- Merge-sort Practice. Refer to the following link. Merge the two sub-arrays into the larger array.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Sorting/MergesortMergePRO.html>

Graphical user interface, text

Description automatically generated

**## Task 2**

- Refer to the following link. Discuss how Quick-sort works:

<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Sorting/quicksortAV.html>

- Quick-sort Practice. Refer to the following link. Partition the array using quicksort.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Sorting/QuicksortPartitPRO.html>

Graphical user interface

Description automatically generated with low confidence

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**## Task 3**

- The following snippet is from `./src/quicksort.cpp` lines 32-43. Discuss in groups how the code works:

```cpp

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        //partition the array

        int pivot = partition(arr, low, high);

        //sort the sub arrays independently

        quickSort(arr, low, pivot - 1);

        quickSort(arr, pivot + 1, high);

    }

}

```

- The following snippet is from `./src/quicksort.cpp` line 27. Discuss in groups how ìt works:

```cpp

swap(&arr[i + 1], &arr[high]);

```

the whole program:

#include <iostream>

using namespace std;

// Swap two elements - Utility function

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

// partition the array using last element as pivot

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];    // pivot

    int i = (low - 1);

    for (int j = low; j <= high- 1; j++)

    {

        //if current element is smaller than pivot, increment the low element

        //swap elements at i and j

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

//quicksort algorithm

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        //partition the array

        int pivot = partition(arr, low, high);

        //sort the sub arrays independently

        quickSort(arr, low, pivot - 1);

        quickSort(arr, pivot + 1, high);

    }

}

void displayArray(int arr[], int size)

{

    int i;

    for (i=0; i < size; i++)

        cout<<arr[i]<<"\t";

}

int main()

{

    int arr[] = {12,23,3,43,51,35,19,45};

    int n = sizeof(arr)/sizeof(arr[0]);

    cout<<"Input array"<<endl;

    displayArray(arr,n);

    cout<<endl;

    quickSort(arr, 0, n-1);

    cout<<"Array sorted with quick sort"<<endl;

    displayArray(arr,n);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture4-main\activity1\src> .\quicksort

Input array

12 23 3 43 51 35 19 45

Array sorted with quick sort

3 12 19 23 35 43 45 51

Quicksort is a fast sorting algorithm that works by splitting a large array of data into smaller sub-arrays. This implies that each iteration works by splitting the input into two components, sorting them, and then recombining them.10.3

Table

Description automatically generated

Time Complexity | Quicksort | Image by Author

**## Task 4: Individual (at home)**

1. Merge-sort has better worst case performance than quicksort. So why Quick-sort is considered better than Merge-sort? Refer to the following article

<https://www.geeksforgeeks.org/quicksort-better-mergesort/>

2. Refer to the following article. Analyze the complexity of the Merge-sort algorithm.

<https://www.softwaretestinghelp.com/merge-sort/>

3. Refer to the following article. Analyze the complexity of the Quick-sort algorithm.

<https://www.softwaretestinghelp.com/quick-sort/>

answers:

**## Task 4: Individual (at home)**

1. Merge-sort has better worst case performance than quicksort. So why Quick-sort is considered better than Merge-sort? Refer to the following article

<https://www.geeksforgeeks.org/quicksort-better-mergesort/>

# Why quicksort is better than mergesort ?

This a common question asked in DS interviews that despite of better worst case performance of mergesort, quicksort is considered better than mergesort. There are certain reasons due to which quicksort is better especially in case of arrays:

1. **Auxiliary Space :**[Mergesort](https://www.geeksforgeeks.org/merge-sort/) uses extra space, [quicksort](https://www.geeksforgeeks.org/quick-sort/) requires little space and exhibits good cache locality. Quick sort is an in-place[sorting algorithm](https://www.geeksforgeeks.org/sorting-algorithms/). In-place sorting means no additional storage space is needed to perform sorting. Merge sort requires a temporary array to merge the sorted arrays and hence it is not in-place giving Quick sort the advantage of space.
2. **Worst Cases :**The worst case of quicksort **O(n2)** can be avoided by using randomized quicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort.
3. **Locality of reference :** Quicksort in particular exhibits good cache locality and this makes it faster than merge sort in many cases like in virtual memory environment.
4. **Merge sort is better for large data structures:**Mergesort is a stable sort, unlike quicksort and heapsort, and can be easily adapted to operate on [linked lists](https://www.geeksforgeeks.org/data-structures/linked-list/) and very large lists stored on slow-to-access media such as disk storage or network attached storage.

2. Refer to the following article. Analyze the complexity of the Merge-sort algorithm.

<https://www.softwaretestinghelp.com/merge-sort/>

**C++ Merge Sort Technique.**

Merge sort algorithm uses the “**divide and conquer**” strategy wherein we divide the problem into subproblems and solve those subproblems individually.

These subproblems are then combined or merged together to form a unified solution

### Overview

**Merge sort is performed using the following steps:**

**#1)** The list to be sorted is divided into two arrays of equal length by dividing the list on the middle element. If the number of elements in the list is either 0 or 1, then the list is considered sorted.

**#2)** Each sublist is sorted individually by using merge sort recursively.

**#3)** The sorted sublists are then combined or merged together to form a complete sorted list.

### General Algorithm

**The general pseudo-code for the merge sort technique is given below.**

Declare an array Arr of length N  
If N=1, Arr is already sorted  
If N>1,  
Left = 0, right = N-1  
Find middle = (left + right)/2  
Call merge\_sort(Arr,left,middle) =>sort first half recursively  
Call merge\_sort(Arr,middle+1,right) => sort second half recursively  
Call merge(Arr, left, middle, right) to merge sorted arrays in above steps.  
Exit

### Complexity Analysis Of The Merge Sort Algorithm

We know that in order to perform sorting using merge sort, we first divide the array into two equal halves. This is represented by “log n” which is a logarithmic function and the number of steps taken is log (n+1) at the most.

Next to find the middle element of the array we require single step i.e. O(1).

Then to merge the sub-arrays into an array of n elements, we will take O (n) amount of running time.

Thus the total time to perform merge sort will be n (log n+1), which gives us the time complexity of O (n\*logn).

|  |  |
| --- | --- |
| Worst case time complexity | O(n\*log n) |
| Best case time complexity | O(n\*log n) |
| Average time complexity | O(n\*log n) |
| Space complexity | O(n) |

The time complexity for merge sort is the same in all three cases (worst, best and average) as it always divides the array into sub-arrays and then merges the sub-arrays taking linear time.

Merge sort always takes an equal amount of space as unsorted arrays. Hence when the list to be sorted is an array, merge sort should not be used for very large arrays. However, merge sort can be used more effectively for linked lists sorting.

### Conclusion

Merge sort uses the “divide and conquer” strategy which divides the array or list into numerous sub arrays and sorts them individually and then merges into a complete sorted array.

Merge sort performs faster than other sorting methods and also works efficiently for smaller and larger arrays likewise.

#include <iostream>

using namespace std;

void merge(int \*,int, int , int );

void merge\_sort(int \*arr, int low, int high)

{

    int mid;

    if (low < high){

        //divide the array at mid and sort independently using merge sort

        mid=(low+high)/2;

        merge\_sort(arr,low,mid);

        merge\_sort(arr,mid+1,high);

        //merge or conquer sorted arrays

        merge(arr,low,high,mid);

    }

}

// Merge sort

void merge(int \*arr, int low, int high, int mid)

{

    int i, j, k, c[50];

    i = low;

    k = low;

    j = mid + 1;

    while (i <= mid && j <= high) {

        if (arr[i] < arr[j]) {

            c[k] = arr[i];

            k++;

            i++;

        }

        else  {

            c[k] = arr[j];

            k++;

            j++;

        }

    }

    while (i <= mid) {

        c[k] = arr[i];

        k++;

        i++;

    }

    while (j <= high) {

        c[k] = arr[j];

        k++;

        j++;

    }

    for (i = low; i < k; i++)  {

        arr[i] = c[i];

    }

}

// read input array and call mergesort

int main()

{

    int myarray[30], num;

    cout<<"Enter number of elements to be sorted:";

    cin>>num;

    cout<<"Enter "<<num<<" elements to be sorted:";

    for (int i = 0; i < num; i++) { cin>>myarray[i];

    }

    merge\_sort(myarray, 0, num-1);

    cout<<"Sorted array\n";

    for (int i = 0; i < num; i++)

    {

        cout<<myarray[i]<<"\t";

    }

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\merge\_sort

Enter number of elements to be sorted:5

Enter 5 elements to be sorted:34

67

899

43

22

Sorted array

22 34 43 67 899

3. Refer to the following article. Analyze the complexity of the Quick-sort algorithm.

<https://www.softwaretestinghelp.com/quick-sort/>

**Quicksort in C++ With Illustration.**

Quicksort is a widely used sorting algorithm which selects a specific element called “pivot” and partitions the array or list to be sorted into two parts based on this pivot s0 that the elements lesser than the pivot are to the left of the list and the elements greater than the pivot are to the right of the list.

Thus the list is partitioned into two sublists. The sublists may not be necessary for the same size. Then Quicksort calls itself recursively to sort these two sublists.

### Introduction

Quicksort works efficiently as well as faster even for larger arrays or lists.

In this tutorial, we will explore more about the working of Quicksort along with some programming examples of the quicksort algorithm.

As a pivot value, we can choose either first, last or the middle value or any random value. The general idea is that ultimately the pivot value is placed at its proper position in the array by moving the other elements in the array to the left or right.

### General Algorithm

**The general algorithm for Quicksort is given below.**

|  |
| --- |
| quicksort(A, low, high)  begin  Declare array A[N] to be sorted      low = 1st element; high = last element; pivot  **if**(low < high)      begin      pivot = partition (A,low,high);      quicksort(A,low,pivot-1)      quicksort(A,pivot+1,high)      End  end |

#include <iostream>

using namespace std;

// Swap two elements - Utility function

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

// partition the array using last element as pivot

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];    // pivot

    int i = (low - 1);

    for (int j = low; j <= high- 1; j++)

    {

        //if current element is smaller than pivot, increment the low element

        //swap elements at i and j

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

//quicksort algorithm

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        //partition the array

        int pivot = partition(arr, low, high);

        //sort the sub arrays independently

        quickSort(arr, low, pivot - 1);

        quickSort(arr, pivot + 1, high);

    }

}

void displayArray(int arr[], int size)

{

    int i;

    for (i=0; i < size; i++)

        cout<<arr[i]<<"\t";

}

int main()

{

    int arr[] = {12,23,3,43,51,35,19,45};

    int n = sizeof(arr)/sizeof(arr[0]);

    cout<<"Input array"<<endl;

    displayArray(arr,n);

    cout<<endl;

    quickSort(arr, 0, n-1);

    cout<<"Array sorted with quick sort"<<endl;

    displayArray(arr,n);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\quick\_sort

Input array

12 23 3 43 51 35 19 45

Array sorted with quick sort

3 12 19 23 35 43 45 51

### Complexity Analysis Of The Quicksort Algorithm

The time taken by quicksort to sort an array depends on the input array and partition strategy or method.

If k is the number of elements less than the pivot and n is the total number of elements, then the general time taken by quicksort can be expressed as follows:

T(n) = T(k) + T(n-k-1) +O (n)

Here, T(k) and T(n-k-1) are the time taken by recursive calls and O(n) is the time taken by partitioning call.

Let us analyze this time complexity for quicksort in detail.

**#1) Worst case**: Worst case in quicksort technique occurs mostly when we select the lowest or highest element in the array as a pivot. (In the above illustration we have selected the highest element as the pivot). In such a situation worst case occurs when the array to be sorted is already sorted in ascending or descending order.

**Hence the above expression for total time taken changes as:**

**T(n) = T(0) + T(n-1) + O(n)**that resolves to**O(n2)**

**#2) Best case:**The best case for quicksort always occurs when the pivot element selected is the middle of the array.

**Thus the recurrence for the best case is:**

**T(n) = 2T(n/2) + O(n) = O(nlogn)**

**#3) Average case:**To analyze the average case for quicksort, we should consider all the array permutations and then calculate the time taken by each of these permutations. In a nutshell, the average time for quicksort also becomes O(nlogn).

**Given below are the various complexities for Quicksort technique:**

|  |  |
| --- | --- |
| Worst case time complexity | O(n 2 ) |
| Best case time complexity | O(n\*log n) |
| Average time complexity | O(n\*log n) |
| Space complexity | O(n\*log n) |

We can implement quicksort in many different ways just by changing the choice of the pivot element (middle, first or last), however, the worst-case rarely occurs for quicksort.

ACTIVITY2

**# Activities**

> The [modulo-calculator](#links) might be handy in these exercises.

**## Task 1**

- Refer to the following link. Discuss how open hashing works.

<https://www.cs.usfca.edu/~galles/visualization/OpenHash.html>

- Open Hashing Practice. Refer to the following link. Move each record on the left to the appropriate bin on the right.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Hashing/OpenHashPRO.html>

- Given the following input (`4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199`) and the hash function `x mod 10`, which of the following statements are true?

- [] `9679, 1989, 4199` hash to the same value

- [] `1471, 6171` hash to the same value

- [] All elements hash to the same value

- [] Each element hashes to a different value

**## Task 2**

- Bucket Hashing Practice. Refer to the following [link](<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Hashing/HashBucketPRO.html>).

- The keys `12, 18, 13, 2, 3, 23, 5 and 15` are inserted into an initially empty hash table of length `10` using open addressing with hash function `h(k) = k mod 10` and **\*\*linear probing\*\***. What is the resultant hash table?

**## Task 3:**

- What is the [Birthday Paradox](<http://en.wikipedia.org/wiki/Birthday_problem>)?

- Why is it generally discussed with hashing?

- In a hash table of 9658 slots, what is the smallest number of records that must be inserted for the probability of a collision to be 61% or more? Use the calculator at this [link](<https://opendsa-server.cs.vt.edu/ODSA/AV/Hashing/Birthday.html>)

- Discuss in groups how the following program works `./src/birthday.cpp`?

**## Task 4: Individual (at home)**

- Difference between `Separate Chaining` and `Open Addressing` collision handling techniques?

<https://www.geeksforgeeks.org/open-addressing-collision-handling-technique-in-hashing/>

<https://www.geeksforgeeks.org/separate-chaining-collision-handling-technique-in-hashing/>

- (Bonus) Run the following program and comment on the code `./src/hashtable.cpp`

**## Link(s)**

- [modulo-calculator](<https://www.calculators.org/math/modulo.php>)

- [Practice Problems on Hashing](<https://www.geeksforgeeks.org/practice-problems-on-hashing/>)

ANSWERS

**## Task 1**

- Refer to the following link. Discuss how open hashing works.

<https://www.cs.usfca.edu/~galles/visualization/OpenHash.html>

- Open Hashing Practice. Refer to the following link. Move each record on the left to the appropriate bin on the right.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Hashing/OpenHashPRO.html>

- Given the following input (`4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199`) and the hash function `x mod 10`, which of the following statements are true?

- [] `9679, 1989, 4199` hash to the same value

- [] `1471, 6171` hash to the same value

- [] All elements hash to the same value

- [] Each element hashes to a different value

Graphical user interface

Description automatically generated

Text

Description automatically generated with medium confidence

- Given the following input (`4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199`) and the hash function `x mod 10`, which of the following statements are true?

- [X] `9679, 1989, 4199` hash to the same value

- [X] `1471, 6171` hash to the same value

- [] All elements hash to the same value

- [] Each element hashes to a different value

**# Task 2**

- Bucket Hashing Practice. Refer to the following [link](<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Hashing/HashBucketPRO.html>).

- The keys `12, 18, 13, 2, 3, 23, 5 and 15` are inserted into an initially empty hash table of length `10` using open addressing with hash function `h(k) = k mod 10` and **\*\*linear probing\*\***. What is the resultant hash table?

Graphical user interface, text

Description automatically generated

- The keys `12, 18, 13, 2, 3, 23, 5 and 15` are inserted into an initially empty hash table of length `10` using open addressing with hash function `h(k) = k mod 10` and **\*\*linear probing\*\***. What is the resultant hash table?

A picture containing text, furniture

Description automatically generated

**## Task 3:**

- What is the [Birthday Paradox](<http://en.wikipedia.org/wiki/Birthday_problem>)?

- Why is it generally discussed with hashing?

- In a hash table of 9658 slots, what is the smallest number of records that must be inserted for the probability of a collision to be 61% or more? Use the calculator at this [link](<https://opendsa-server.cs.vt.edu/ODSA/AV/Hashing/Birthday.html>)

- Discuss in groups how the following program works `./src/birthday.cpp`?

ANSWERS:

- What is the [Birthday Paradox](<http://en.wikipedia.org/wiki/Birthday_problem>)?

- Why is it generally discussed with hashing?

In [probability theory](https://en.wikipedia.org/wiki/Probability_theory), the **birthday problem** asks for the probability that, in a set of *n* [randomly](https://en.wikipedia.org/wiki/Random) chosen people, at least two will share a [birthday](https://en.wikipedia.org/wiki/Birthday). The **birthday paradox** refers to the counterintuitive fact that only 23 people are needed for that probability to exceed 50%

Real-world applications for the birthday problem include a cryptographic attack called the [birthday attack](https://en.wikipedia.org/wiki/Birthday_attack), which uses this probabilistic model to reduce the complexity of finding a [collision](https://en.wikipedia.org/wiki/Collision_attack) for a [hash function](https://en.wikipedia.org/wiki/Hash_function), as well as calculating the approximate risk of a hash collision existing within the hashes of a given size of population.

- In a hash table of 9658 slots, what is the smallest number of records that must be inserted for the probability of a collision to be 61% or more? Use the calculator at this [link](<https://opendsa-server.cs.vt.edu/ODSA/AV/Hashing/Birthday.html>)

135

Graphical user interface, text

Description automatically generated

- Discuss in groups how the following program works `./src/birthday.cpp`?

// https : // www.geeksforgeeks.org/birthday-paradox/

// C++ program to approximate number of people in Birthday Paradox

// problem

#include <cmath>

#include <iostream>

using namespace std;

// Returns approximate number of people for a given probability

int find(double p)

{

    return ceil(sqrt(2 \* 365 \* log(1 / (1 - p))));

}

int main()

{

    cout << find(0.70);

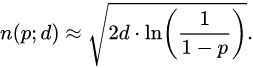
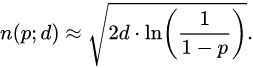
}

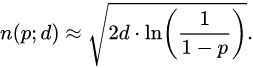
PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture4-main\activity2\src> .\birthday

30

So you need only 30 persons and you get 70 percent probability that somebody has the same birthday.

Conversely, if *n*(*p*; *d*) denotes the number of random integers drawn from [1,*d*] to obtain a probability *p* that at least two numbers are the same, then

�(�;�)≈2�⋅ln⁡(11−�).



The birthday problem in this more generic sense applies to [hash functions](https://en.wikipedia.org/wiki/Hash_function): the expected number of *N*-[bit](https://en.wikipedia.org/wiki/Bit) hashes that can be generated before getting a collision is not 2*N*, but rather only 2*N*⁄2. This is exploited by [birthday attacks](https://en.wikipedia.org/wiki/Birthday_attack) on [cryptographic hash functions](https://en.wikipedia.org/wiki/Cryptographic_hash_function) and is the reason why a small number of collisions in a [hash table](https://en.wikipedia.org/wiki/Hash_table) are, for all practical purposes, inevitable.

The theory behind the birthday problem was used by Zoe Schnabel[[17]](https://en.wikipedia.org/wiki/Birthday_problem#cite_note-18) under the name of [capture-recapture](https://en.wikipedia.org/wiki/Mark_and_recapture) statistics to estimate the size of fish population in lakes.

Conversely, if *n*(*p*; *d*) denotes the number of random integers drawn from [1,*d*] to obtain a probability *p* that at least two numbers are the same, then

�(�;�)≈2�⋅ln⁡(11−�).

The birthday problem in this more generic sense applies to [hash functions](https://en.wikipedia.org/wiki/Hash_function): the expected number of *N*-[bit](https://en.wikipedia.org/wiki/Bit) hashes that can be generated before getting a collision is not 2*N*, but rather only 2*N*⁄2. This is exploited by [birthday attacks](https://en.wikipedia.org/wiki/Birthday_attack) on [cryptographic hash functions](https://en.wikipedia.org/wiki/Cryptographic_hash_function) and is the reason why a small number of collisions in a [hash table](https://en.wikipedia.org/wiki/Hash_table) are, for all practical purposes, inevitable.

The theory behind the birthday problem was used by Zoe Schnabel[[17]](https://en.wikipedia.org/wiki/Birthday_problem#cite_note-18) under the name of [capture-recapture](https://en.wikipedia.org/wiki/Mark_and_recapture) statistics to estimate the size of fish population in lakes.

**## Task 4: Individual (at home)**

- Difference between `Separate Chaining` and `Open Addressing` collision handling techniques?

<https://www.geeksforgeeks.org/open-addressing-collision-handling-technique-in-hashing/>

<https://www.geeksforgeeks.org/separate-chaining-collision-handling-technique-in-hashing/>

- (Bonus) Run the following program and comment on the code `./src/hashtable.cpp`

**## Link(s)**

- [modulo-calculator](<https://www.calculators.org/math/modulo.php>)

- [Practice Problems on Hashing](<https://www.geeksforgeeks.org/practice-problems-on-hashing/>)

**Comparison of the above three:**

* Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
* Quadratic probing lies between the two in terms of cache performance and clustering.
* Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

| **S.No.** | **Separate Chaining** | **Open Addressing** |
| --- | --- | --- |
| 1. | Chaining is Simpler to implement. | Open Addressing requires more computation. |
| 2. | In chaining, Hash table never fills up, we can always add more elements to chain. | In open addressing, table may become full. |
| 3. | Chaining is Less sensitive to the hash function or load factors. | Open addressing requires extra care to avoid clustering and load factor. |
| 4. | Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted. | Open addressing is used when the frequency and number of keys is known. |
| 5. | Cache performance of chaining is not good as keys are stored using linked list. | Open addressing provides better cache performance as everything is stored in the same table. |
| 6. | Wastage of Space (Some Parts of hash table in chaining are never used). | In Open addressing, a slot can be used even if an input doesn’t map to it. |
| 7. | Chaining uses extra space for links. | No links in Open addressing |

**Note:** Cache performance of chaining is not good because when we traverse a Linked List, we are basically jumping from one node to another, all across the computer’s memory. For this reason, the CPU cannot cache the nodes which aren’t visited yet, this doesn’t help us. But with Open Addressing, data isn’t spread, so if the CPU detects that a segment of memory is constantly being accessed, it gets cached for quick access.

**Performance of Open Addressing:**

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

*m = Number of slots in the hash table*

*n = Number of keys to be inserted in the hash table*

*Load factor α = n/m  ( < 1 )*

*Expected time to search/insert/delete < 1/(1 – α)*

*So Search, Insert and Delete take (1/(1 – α)) time*

## ****What is Collision?****

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

## ****What are the chances of collisions with the large table?****

Collisions are very likely even if we have a big table to store keys. An important observation is [Birthday Paradox](https://www.geeksforgeeks.org/birthday-paradox/). With only 23 persons, the probability that two people have the same birthday is 50%.

## ****How to handle Collisions?****

There are mainly two methods to handle collision:

* Separate Chaining
* Open Addressing

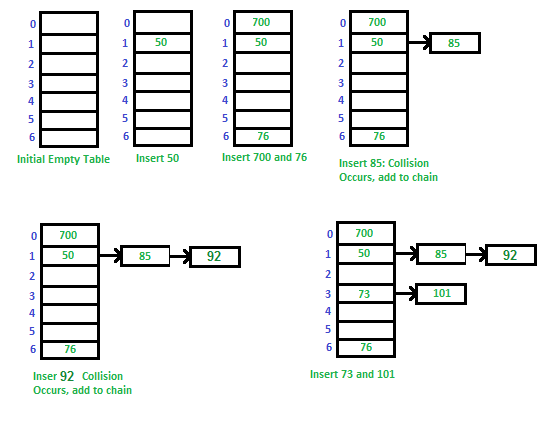
## ****Separate Chaining:****

The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

*The****linked list****data structure is used to implement this technique. So what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*

Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven’t found our entry then it means that the entry does not exist. Hence, the conclusion is that in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

**Example:** Let us consider a simple hash function as “**key mod 7**” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101



hastable.cpp

#include <iostream>

#include <list>

using namespace std;

class HashMapTable

{

    // size of the hash table

    int table\_size;

    // Pointer to an array containing the keys

    list<int> \*table;

public:

    // creating constructor of the above class containing all the methods

    HashMapTable(int key);

    // hash function to compute the index using table\_size and key

    int hashFunction(int key)

    {

        return (key % table\_size);

    }

    // inserting the key in the hash table

    void insertElement(int key);

    // deleting the key in the hash table

    void deleteElement(int key);

    // displaying the full hash table

    void displayHashTable();

};

// creating the hash table with the given table size

HashMapTable::HashMapTable(int ts)

{

    this->table\_size = ts;

    table = new list<int>[table\_size];

}

// insert function to push the keys in hash table

void HashMapTable::insertElement(int key)

{

    int index = hashFunction(key);

    table[index].push\_back(key);

}

// delete function to delete the element from the hash table

void HashMapTable::deleteElement(int key)

{

    int index = hashFunction(key);

    // finding the key at the computed index

    list<int>::iterator i;

    for (i = table[index].begin(); i != table[index].end(); i++)

    {

        if (\*i == key)

            break;

    }

    // removing the key from hash table if found

    if (i != table[index].end())

        table[index].erase(i);

}

// display function to showcase the whole hash table

void HashMapTable::displayHashTable()

{

    for (int i = 0; i < table\_size; i++)

    {

        cout << i;

        // traversing at the recent/ current index

        for (auto j : table[i])

            cout << " ==> " << j;

        cout << endl;

    }

}

// Main function

int main()

{

    // array of all the keys to be inserted in hash table

    int arr[] = {20, 34, 56, 54, 76, 87};

    int n = sizeof(arr) / sizeof(arr[0]);

    // table\_size of hash table as 6

    HashMapTable ht(6);

    for (int i = 0; i < n; i++)

        ht.insertElement(arr[i]);

    // deleting element 34 from the hash table

    ht.deleteElement(34);

    // displaying the final data of hash table

    ht.displayHashTable();

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture4-main\activity2\src> .\hashtable

0 ==> 54

1

2 ==> 20 ==> 56

3 ==> 87

4 ==> 76

5

<https://www.geeksforgeeks.org/c-program-hashing-chaining/>

// CPP program to implement hashing with chaining

#include<bits/stdc++.h>

using namespace std;

class Hash

{

    int BUCKET;    // No. of buckets

    // Pointer to an array containing buckets

    list<int> \*table;

public:

    Hash(int V);  // Constructor

    // inserts a key into hash table

    void insertItem(int x);

    // deletes a key from hash table

    void deleteItem(int key);

    // hash function to map values to key

    int hashFunction(int x) {

        return (x % BUCKET);

    }

    void displayHash();

};

Hash::Hash(int b)

{

    this->BUCKET = b;

    table = new list<int>[BUCKET];

}

void Hash::insertItem(int key)

{

    int index = hashFunction(key);

    table[index].push\_back(key);

}

void Hash::deleteItem(int key)

{

  // get the hash index of key

  int index = hashFunction(key);

  // find the key in (index)th list

  list <int> :: iterator i;

  for (i = table[index].begin();

           i != table[index].end(); i++) {

    if (\*i == key)

      break;

  }

  // if key is found in hash table, remove it

  if (i != table[index].end())

    table[index].erase(i);

}

// function to display hash table

void Hash::displayHash() {

  for (int i = 0; i < BUCKET; i++) {

    cout << i;

    for (auto x : table[i])

      cout << " --> " << x;

    cout << endl;

  }

}

// Driver program

int main()

{

  // array that contains keys to be mapped

  int a[] = {15, 11, 27, 8, 12};

  int n = sizeof(a)/sizeof(a[0]);

  // insert the keys into the hash table

  Hash h(7);   // 7 is count of buckets in

               // hash table

  for (int i = 0; i < n; i++)

    h.insertItem(a[i]);

  // delete 12 from hash table

  h.deleteItem(12);

  // display the Hash table

  h.displayHash();

  return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\hashing\_with\_chaining

0

1 --> 15 --> 8

2

3

4 --> 11

5

6 --> 27

ACTIVITY3

**# Activities**

**## Task 1:**

- Refer to the following link. Discuss how Stacks based on linked lists works:

<https://www.cs.usfca.edu/~galles/visualization/StackLL.html>

**## Task 2:**

The following snippet is from `./src/stack.cpp` lines 25-31.

```cpp

void push(StackNode \*\*root, int new\_data)

{

    StackNode \*stackNode = newNode(new\_data);

    stackNode->next = \*root;

    \*root = stackNode;

    cout << new\_data << endl;

}

```

- Discuss in groups how the code works?

- Why do we need to use double pointers?

**## Task 3: Individual (At home)**

Practice: Asymptotic Analysis and Upper Bounds. Refer to the following link:

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/AlgAnal/UpperBoundsSumm.html>

ANSWERS:

**## Task 1:**

- Refer to the following link. Discuss how Stacks based on linked lists works:

<https://www.cs.usfca.edu/~galles/visualization/StackLL.html>

A picture containing text

Description automatically generated

<https://www.geeksforgeeks.org/implement-a-stack-using-singly-linked-list/>

**Stack Operations:**

* [**push()**](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/)**:** Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
* [**pop()**](https://www.geeksforgeeks.org/stack-push-and-pop-in-c-stl/)**:** Return the top element of the Stack i.e simply delete the first element from the linked list.
* [**peek()**](https://www.geeksforgeeks.org/stack-peek-method-in-java/)**:** Return the top element.
* **display():** Print all elements in Stack.

**## Task 2:**

The following snippet is from `./src/stack.cpp` lines 25-31.

```cpp

void push(StackNode \*\*root, int new\_data)

{

    StackNode \*stackNode = newNode(new\_data);

    stackNode->next = \*root;

    \*root = stackNode;

    cout << new\_data << endl;

}

```

- Discuss in groups how the code works?

- Why do we need to use double pointers?

whole code:

#include <iostream>

using namespace std;

// class to represent a stack node

class StackNode

{

public:

    int data;

    StackNode \*next;

};

StackNode \*newNode(int data)

{

    StackNode \*stackNode = new StackNode();

    stackNode->data = data;

    stackNode->next = NULL;

    return stackNode;

}

int isEmpty(StackNode \*root)

{

    return !root;

}

void push(StackNode \*\*root, int new\_data)

{

    StackNode \*stackNode = newNode(new\_data);

    stackNode->next = \*root;

    \*root = stackNode;

    cout << new\_data << endl;

}

int pop(StackNode \*\*root)

{

    if (isEmpty(\*root))

        return -1;

    StackNode \*temp = \*root;

    \*root = (\*root)->next;

    int popped = temp->data;

    free(temp);

    return popped;

}

int peek(StackNode \*root)

{

    if (isEmpty(root))

        return -1;

    return root->data;

}

int main()

{

    StackNode \*root = NULL;

    cout << "Stack Push:" << endl;

    push(&root, 100);

    push(&root, 200);

    push(&root, 300);

    cout << "\nTop element is " << peek(root) << endl;

    cout << "\nStack Pop:" << endl;

    while (!isEmpty(root))

    {

        cout << pop(&root) << endl;

    }

    cout << "Top element is " << peek(root) << endl;

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture4-main\activity3\src> .\stack

Stack Push:

100

200

300

Top element is 300

Stack Pop:

300

200

100

Top element is -1

<https://www.softwaretestinghelp.com/stack-in-cpp/>

**All That You Need To Know About Stack In C++.**

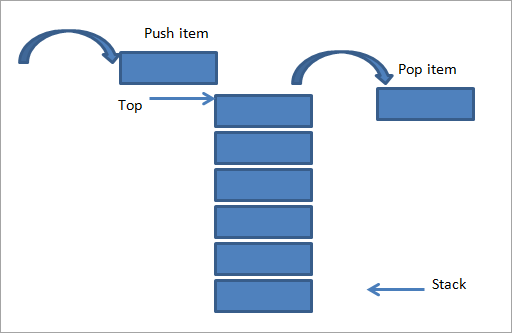
Stack is a fundamental data structure which is used to store elements in a linear fashion.

Stack follows **LIFO (last in, first out)** order or approach in which the operations are performed. This means that the element which was added last to the stack will be the first element to be removed from the stack.

## Stack In C++

A stack is similar to real-life stack or a pile of things that we stack one above the other.

**Given below is a pictorial representation of Stack.**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/pictorial-representation-of-stack.png)

As shown above, there is a pile of plates stacked on top of each other. If we want to add another item to it, then we add it at the top of the stack as shown in the above figure (left-hand side). This operation of adding an item to stack is called “**Push**”.

On the right side, we have shown an opposite operation i.e. we remove an item from the stack. This is also done from the same end i.e. the top of the stack. This operation is called “**Pop**”.

As shown in the above figure, we see that push and pop are carried out from the same end. This makes the stack to follow LIFO order. The position or end from which the items are pushed in or popped out to/from the stack is called the “**Top of the stack**”.

Initially, when there are no items in the stack, the top of the stack is set to -1. When we add an item to the stack, the top of the stack is incremented by 1 indicating that the item is added. As opposed to this, the top of the stack is decremented by 1 when an item is popped out of the stack.

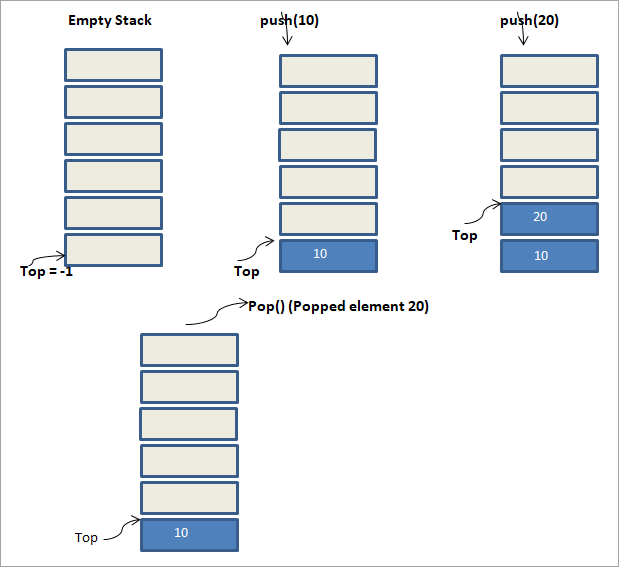
Next, we will see some of the basic operations of the stack data structure that we will require while implementing the stack.

### Basic Operations

Following are the basic operations that are supported by the stack.

* **push –** Adds or pushes an element into the stack.
* **pop –**Removes or pops an element out of the stack.
* **peek –** Gets the top element of the stack but doesn’t remove it.
* **isFull –**Tests if the stack is full.
* **isEmpty –** Tests if the stack is empty.

### Illustration

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/Stack-Illustration.png)

The above illustration shows the sequence of operations that are performed on the stack. Initially, the stack is empty. For an empty stack, the top of the stack is set to -1.

Next, we push the element 10 into the stack. We see that the top of the stack now points to element 10.

Next, we perform another push operation with element 20, as a result of which the top of the stack now points to 20. This state is the third figure.

Now in the last figure, we perform a pop () operation. As a result of the pop operation, the element pointed at the top of the stack is removed from the stack. Hence in the figure, we see that element 20 is removed from the stack. Thus the top of the stack now points to 10.

In this way, we can easily make out the LIFO approach used by stack.

- Why do we need to use double pointers?

If it is single pointer, the data structure is updated but caller won't see it.

**## Task 3: Individual (At home)**

Practice: Asymptotic Analysis and Upper Bounds. Refer to the following link:

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/AlgAnal/UpperBoundsSumm.html>

Graphical user interface, text, application, email

Description automatically generated

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/AlgAnal/UpperBoundsSumm.html>

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated