**Outline**

**Topics**

* Review
* Algorithm techniques:
  + Brute Force
  + Decrease-and-Conquer
* Analyses tool(s): Big �

**Part 0: Review**

* Algorithm techniques:
  + Brute Force: [Linear search](https://www.cs.usfca.edu/~galles/visualization/Search.html)
  + Decrease-and-Conquer: [Binary Search](https://www.cs.usfca.edu/~galles/visualization/Search.html)
  + Divide-and-Conquer
* [Big $O$](https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/)
* [Recursion](https://recursion.vercel.app/) vs Iteration

**Grading/ Week (Points)**

* Week 1 (Max 10 points)
* Week 2 (Max 15 points)
* Week 3
  + Group Activities (Max 9 points)
  + Homework (Individual activities) (Max 9 points)
  + [Peer reviews (Max 7 points)](https://github.com/TT00FE39-3001/lecture3/blob/main/peer-assessment-template.xlsx)

**Part 1: Sorting using brute force**

* Algorithm techniques: **Brute Force**
  + Bubble sort
  + Selection sort
  + Big � analysis
* [Activity 1](https://github.com/TT00FE39-3001/lecture3/blob/main/activity1/README.md)

**Part 2: Sorting using decrease and conquer**

* Algorithm techniques: **decrease and conquer**
  + Insertion sort
  + Big � analysis
  + Average case vs worst case
* [Activity 2](https://github.com/TT00FE39-3001/lecture3/blob/main/activity2/README.md)

**Part 3: Introduction to Linked list**

* [Big O Complexity](https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1176/handouts/midterm/5-BigO.pdf)
* Introduction to linked list
  + Queues
  + Stack
* [Activity 3](https://github.com/TT00FE39-3001/lecture3/blob/main/activity3/README.md)

**Misc**

* [Links](https://github.com/TT00FE39-3001/lecture3/blob/main/links.md)
* [Cheat Sheet: Mathematical Notation in Markdown](https://upyesp.org/posts/makrdown-vscode-math-notation/)

**About**

*No description, website, or topics provided.*

**Resources**

[Readme](https://github.com/TT00FE39-3001/lecture3" \l "readme)

**Stars**

**[0](https://github.com/TT00FE39-3001/lecture3/stargazers)**[stars](https://github.com/TT00FE39-3001/lecture3/stargazers)

**Watchers**

**[0](https://github.com/TT00FE39-3001/lecture3/watchers)**[watching](https://github.com/TT00FE39-3001/lecture3/watchers)

**Forks**

**[0](https://github.com/TT00FE39-3001/lecture3/network/members)**[forks](https://github.com/TT00FE39-3001/lecture3/network/members)

**[Releases](https://github.com/TT00FE39-3001/lecture3/releases)**

No releases published

[**Packages**](https://github.com/orgs/TT00FE39-3001/packages?repo_name=lecture3)

No packages published

**Languages**

* [C++89.1%](https://github.com/TT00FE39-3001/lecture3/search?l=c%2B%2B)
* [Python10.9%](https://github.com/TT00FE39-3001/lecture3/search?l=python)

**Footer**

© 2023 GitHub, Inc.

**Footer navigation**

* [Terms](https://docs.github.com/site-policy/github-terms/github-terms-of-service)
* [Privacy](https://docs.github.com/site-policy/privacy-policies/github-privacy-statement)
* [Security](https://github.com/security)
* [Status](https://www.githubstatus.com/)
* [D](https://docs.github.com/)

v**# Outline**

**## Topics**

- Review

- Algorithm techniques:

  - Brute Force

  - Decrease-and-Conquer

- Analyses tool(s): Big $O$

---

**## Part 0: Review**

- Algorithm techniques:

  - Brute Force: [Linear search](<https://www.cs.usfca.edu/~galles/visualization/Search.html>)

  - Decrease-and-Conquer: [Binary Search](<https://www.cs.usfca.edu/~galles/visualization/Search.html>)

  - Divide-and-Conquer

- [Big $O$](<https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/>)

- [Recursion](<https://recursion.vercel.app/>)

vs Iteration

**## Grading/ Week (Points)**

- Week 1 (Max 10 points)

- Week 2 (Max 15 points)

- Week 3

  - Group Activities (Max 9 points)

  - Homework (Individual activities) (Max 9 points)

  - [Peer reviews (Max 7 points)](./peer-assessment-template.xlsx)

---

**## Part 1: Sorting using brute force**

- Algorithm techniques: **\*\*Brute Force\*\***

  - Bubble sort

  - Selection sort

  - Big $O$ analysis

- [Activity 1](./activity1/README.md)

<!-- average case vs worst case -->

---

**## Part 2: Sorting using decrease and conquer**

- Algorithm techniques: **\*\*decrease and conquer\*\***

  - Insertion sort

  - Big $O$ analysis

  - Average case vs worst case

- [Activity 2](./activity2/README.md)

---

**## Part 3: Introduction to Linked list**

- [Big O Complexity](<https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1176/handouts/midterm/5-BigO.pdf>)

- Introduction to linked list

  - Queues

  - Stack

- [Activity 3](./activity3/README.md)

---

**## Misc**

- [Links](./links.md)

- [Cheat Sheet: Mathematical Notation in Markdown](<https://upyesp.org/posts/makrdown-vscode-math-notation/>)

links:

**# Links**

- [Data Structures In C++](<https://www.softwaretestinghelp.com/cpp-tutorials/>)

- [Visualization of Algorithms](<https://www.cs.usfca.edu/~galles/visualization>)

- [Data Structures and Algorithms](<https://www.techiedelight.com/>)

- [Understanding the formal definition of Big-O](<https://justin.abrah.ms/computer-science/understanding-big-o-formal-definition.html>)

- Bubble Sort

  - [Bubble-sort in C++](<https://www.softwaretestinghelp.com/bubble-sort/>)

  - [Bubble-sort Visualizer](<https://opendsa-server.cs.vt.edu/embed/bubblesortAV>)

  - [Bubble-sort tutorial](<https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BubbleSort.html>)

- Selection Sort

  - [Selection-sort Visualizer](<https://opendsa-server.cs.vt.edu/ODSA/AV/Sorting/selectionsortAV.html>)

  - [Selection-sort In C++](<https://www.softwaretestinghelp.com/selection-sort/>)

  - [Selection-sort tutorial](<https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/SelectionSort.html>)

- [OpenDSA Data Structures and Algorithms Modules Collection Table Of Contents](<https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/index.html>)

- [Math support in Markdown](<https://github.blog/2022-05-19-math-support-in-markdown/>)

- [Cheat Sheet: Mathematical Notation in Markdown](<https://upyesp.org/posts/makrdown-vscode-math-notation/>)

- [Visualization: Queue based on linked list](<https://www.cs.usfca.edu/~galles/visualization/QueueLL.html>)

- [Visualization: Stack based on linked list](<https://www.cs.usfca.edu/~galles/visualization/StackLL.html>)

- <https://cpp.sh/>

HOMEWORK

**# Homework**

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

- [Sorts 1 Introduction to sorts (~8min)](<https://www.youtube.com/watch?v=H3FCoYQMKvI>)

- [Sorts 2 Selection Sort (~5min)](<https://youtu.be/fgYlVyrt1vE>)

- [Sorts 3 Insertion Sort (~5min)](<https://youtu.be/eTvQIbB-AuE>)

- [Sorts 4 Insertion Sort Code (~4min)](<https://youtu.be/3U2vTqaL7uE>)

-

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

- [Bubble Sort In C++ With Examples](https://www.softwaretestinghelp.com/bubble-sort/)

- [Selection Sort In C++ With Examples](https://www.softwaretestinghelp.com/selection-sort/)

- [Insertion Sort In C++ With Examples](https://www.softwaretestinghelp.com/insertion-sort/)

**## Task 3/3: Pre-Lecture**

- [Quicksort](https://youtu.be/0SkOjNaO1XY)

- [Pointer to pointer in c++](https://youtu.be/d3kd5KbGB48)

- [Hash Tables and Dictionaries](https://youtu.be/sfWyugl4JWA)

**## Recommended**

- [C++ POINTERS FULL COURSE Beginner to Advanced (150min)](<https://youtu.be/kiUGf_Z08RQ>)

ACTIVITY1

**# Activities**

**## Task 1**

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

<https://opendsa-server.cs.vt.edu/embed/bubblesortAV>

**## Task 2**

- Refer to the following link. Your task is to show the behavior for one iteration of the outer for loop of Bubble Sort (Try at least 3 cases).

<https://opendsa-server.cs.vt.edu/ODSA/Exercises/Sorting/BubsortPRO.html>

**## Task 3**

- The following snippet is from `./src/bubble.cpp` lines 16-28. Discuss in groups how the code works:

```cpp

    for (i = 0; i < 10; i++)

    {

        for (j = i + 1; j < 10; j++)

        {

            if (a[j] < a[i])

            {

                temp = a[i];

                a[i] = a[j];

                a[j] = temp;

            }

        }

        pass++;

    }

```

- The following snippet is from `./src/selection.cpp` lines 34-41. Discuss in groups how the code works:

```cpp

    for (j = i + 1; j < 10; j++)

    {

        if (myarray[j] < ele\_small)

        {

            ele\_small = myarray[j];

            position = j;

        }

    }

```

**## Task 4: Individual, at home**

- Discuss the complexity analysis of selection sort. Refer to the link below:

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

**## Links**

- <https://cpp.sh/>

ANSWERS:

**## Task 1**

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

<https://opendsa-server.cs.vt.edu/embed/bubblesortAV>

Graphical user interface, application

Description automatically generated

Bubble sort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. This passing procedure is repeated until no swaps are required, indicating that the list is sorted

**## Task 2**

- Refer to the following link. Your task is to show the behavior for one iteration of the outer for loop of Bubble Sort (Try at least 3 cases).

<https://opendsa-server.cs.vt.edu/ODSA/Exercises/Sorting/BubsortPRO.html>

Move from left to right through the array.

**At each position, if the value is greater than the value to its right, then swap them.**

Graphical user interface, text, application, email

Description automatically generated

**## Task 3**

- The following snippet is from `./src/bubble.cpp` lines 16-28. Discuss in groups how the code works:

```cpp

    for (i = 0; i < 10; i++)

    {

        for (j = i + 1; j < 10; j++)

        {

            if (a[j] < a[i])

            {

                temp = a[i];

                a[i] = a[j];

                a[j] = temp;

            }

        }

        pass++;

    }

```

- The following snippet is from `./src/selection.cpp` lines 34-41. Discuss in groups how the code works:

```cpp

    for (j = i + 1; j < 10; j++)

    {

        if (myarray[j] < ele\_small)

        {

            ele\_small = myarray[j];

            position = j;

        }

    }

```

ANSWERS

#include <iostream>

using namespace std;

int main()

{

    int i, j, temp, pass = 0;

    int a[10] = {10, 2, 0, 14, 43, 25, 18, 1, 5, 45};

    cout << "Input list ...\n";

    for (i = 0; i < 10; i++)

    {

        cout << a[i] << "\t";

    }

    cout << endl;

    for (i = 0; i < 10; i++)

    {

        for (j = i + 1; j < 10; j++)

        {

            if (a[j] < a[i])

            {

                temp = a[i];

                a[i] = a[j];

                a[j] = temp;

            }

        }

        pass++;

    }

    cout << "Sorted Element List ...\n";

    for (i = 0; i < 10; i++)

    {

        cout << a[i] << "\t";

    }

    cout << "\nNumber of passes taken to sort the list:" << pass << endl;

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity1\src> .\bubble

Input list ...

10 2 0 14 43 25 18 1 5 45

Sorted Element List ...

0 1 2 5 10 14 18 25 43 45

Number of passes taken to sort the list:10

#include <iostream>

using namespace std;

int findSmallest(int[], int);

int main()

{

    int myarray[10] = {11, 5, 2, 20, 42, 53, 23, 34, 101, 22};

    int pos, temp, pass = 0;

    cout << "\n Input list of elements to be Sorted\n";

    for (int i = 0; i < 10; i++)

    {

        cout << myarray[i] << "\t";

    }

    for (int i = 0; i < 10; i++)

    {

        pos = findSmallest(myarray, i);

        temp = myarray[i];

        myarray[i] = myarray[pos];

        myarray[pos] = temp;

        pass++;

    }

    cout << "\n Sorted list of elements is\n";

    for (int i = 0; i < 10; i++)

    {

        cout << myarray[i] << "\t";

    }

    cout << "\nNumber of passes required to sort the array: " << pass;

    return 0;

}

int findSmallest(int myarray[], int i)

{

    int ele\_small, position, j;

    ele\_small = myarray[i];

    position = i;

    for (j = i + 1; j < 10; j++)

    {

        if (myarray[j] < ele\_small)

        {

            ele\_small = myarray[j];

            position = j;

        }

    }

    return position;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity1\src> .\selection

Input list of elements to be Sorted

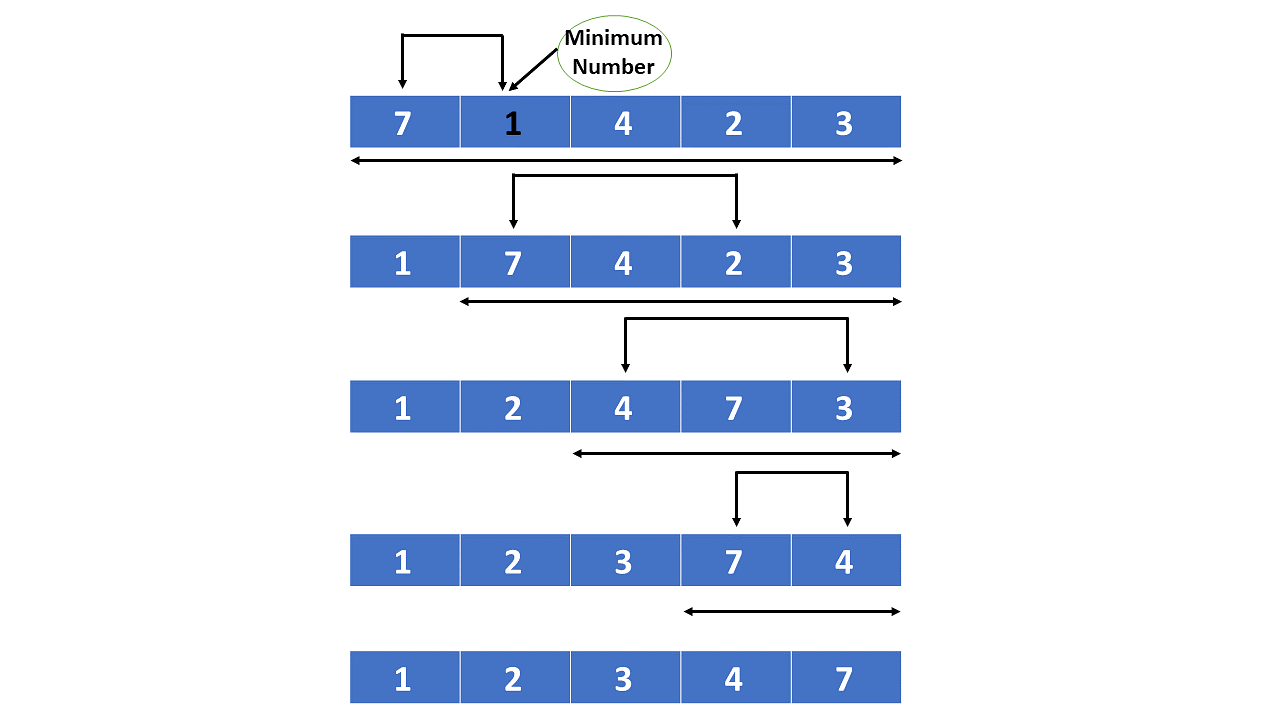
11 5 2 20 42 53 23 34 101 22

Sorted list of elements is

2 5 11 20 22 23 34 42 53 101

Number of passes required to sort the array: 10

Selection sort is an effective and efficient sort algorithm based on comparison operations. It adds one element in each iteration. You need to select the smallest element in the array and move it to the beginning of the array by swapping with the front element.



**## Task 4: Individual, at home**

- Discuss the complexity analysis of selection sort. Refer to the link below:

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

**## Links**

- <https://cpp.sh/>

**An In-Depth Look At Selection Sort In C++ With Examples.**

As the name itself suggests, the selection sort technique first selects the smallest element in the array and swaps it with the first element in the array.

Next, it swaps the second smallest element in the array with the second element and so on. Thus for every pass, the smallest element in the array is selected and put in its proper position until the entire array is sorted.

**Introduction**

Selection sort is quite a straightforward sorting technique as the technique only involves finding the smallest element in every pass and placing it in the correct position.

Selection sort works efficiently when the list to be sorted is of small size but its performance is affected badly as the list to be sorted grows in size.

Hence we can say that selection sort is not advisable for larger lists of data.

**General Algorithm**

The General Algorithm for selection sort is given below:

**Selection Sort (A, N)**

**Step 1**: Repeat Steps 2 and 3 for K = 1 to N-1  
**Step 2**: Call routine smallest(A, K, N,POS)  
**Step 3**: Swap A[K] with A [POS]  
[End of loop]  
**Step 4**: EXIT

**Routine smallest (A, K, N, POS)**

* **Step 1**: [initialize] set smallestElem = A[K]
* **Step 2**: [initialize] set POS = K
* **Step 3**: for J = K+1 to N -1,repeat  
  if smallestElem > A [J]  
  set smallestElem = A [J]  
  set POS = J  
  [if end]  
  [End of loop]
* **Step 4**: return POS

**Pseudocode For Selection Sort**

|  |
| --- |
| Procedure selection\_sort(array,N)      array – array of items to be sorted      N – size of array  begin  **for** I = 1 to N-1      begin          set min  = i  **for** j = i+1 to N          begin  **if** array[j] < array[min] then                  min = j;              end **if**          end **for**          //swap the minimum element with current element  **if** minIndex != I then              swap array[min[] and array[i]          end **if**      end **for**  end procedure |

An example to illustrate this selection sort algorithm is shown below.

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

Input list of elements to be Sorted

11 5 2 20 42 53 23 34 101 22

Sorted list of elements is

2 5 11 20 22 23 34 42 53 101

Number of passes required to sort the array: 10

**Complexity Analysis Of Selection Sort**

As seen in the pseudocode above for selection sort, we know that selection sort requires two for loops nested with each other to complete itself. One for loop steps through all the elements in the array and we find the minimum element index using another for loop which is nested inside the outer for loop.

Therefore, given a size N of the input array, the selection sort algorithm has the following time and complexity values.

|  |  |
| --- | --- |
| Worst case time complexity | O( n 2 ) ; O(n) swaps |
| Best case time complexity | O( n 2 ) ; O(n) swaps |
| Average time complexity | O( n 2 ) ; O(n) swaps |
| Space complexity | O(1) |

The time complexity of O(n2) is mainly because of the use of two for loops. Note that the selection sort technique never takes more than O(n) swaps and is beneficial when the memory write operation proves to be costly.

### Conclusion

Selection sort is yet another simplest sorting technique that can be easily implemented. Selection sort works best when the range of the values to be sorted is known. Thus as far as sorting of data structures using selection sort is concerned, we can only sort data structure which are linear and of finite size.

This means that we can efficiently sort data structures like arrays using the selection sort.

In this tutorial, we have discussed selection sort in detail including the implementation of selection sort using C++ and Java languages. The logic behind the selection sort is to find the smallest element in the list repeatedly and place it in the proper position.

In the next tutorial, we will learn in detail about insertion sort which is said to be a more efficient technique than the other two techniques that we have discussed so far i.e. bubble sort and selection sort.

ACTIVITY 2

**# Activities**

**## Task 1**

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

<https://opendsa-server.cs.vt.edu/ODSA/AV/Sorting/insertionsortAV.html>

**## Task 2**

- The following snippet is from `./src/insertion.cpp` lines 12-22. Discuss in groups how the code works:

```cpp

    for (int k = 1; k < 10; k++)

    {

        int temp = myarray[k];

        int j = k - 1;

        while (j >= 0 && temp <= myarray[j])

        {

            myarray[j + 1] = myarray[j];

            j = j - 1;

        }

        myarray[j + 1] = temp;

    }

```

**## Task 3**

- Discuss the complexity analysis of insertion sort. Refer to the link below:

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

**## Links**

- <https://cpp.sh/>

ANSWERS:

**## Task 1**

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

<https://opendsa-server.cs.vt.edu/ODSA/AV/Sorting/insertionsortAV.html>

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

Graphical user interface, application

Description automatically generated



**Pseudo Code**

procedure insertionSort(A: list of sortable items)

n = length(A)

for i = 1 to n - 1 do

j = i

while j > 0 and A[j-1] > A[j] do

swap(A[j], A[j-1])

j = j - 1

end while

end for

end procedure

// C++ program for insertion sort

#include <bits/stdc++.h>

using namespace std;

// Function to sort an array using

// insertion sort

void insertionSort(int arr[], int n)

{

    int i, key, j;

    for (i = 1; i < n; i++)

    {

        key = arr[i];

        j = i - 1;

        // Move elements of arr[0..i-1],

        // that are greater than key, to one

        // position ahead of their

        // current position

        while (j >= 0 && arr[j] > key)

        {

            arr[j + 1] = arr[j];

            j = j - 1;

        }

        arr[j + 1] = key;

    }

}

// A utility function to print an array

// of size n

void printArray(int arr[], int n)

{

    int i;

    for (i = 0; i < n; i++)

        cout << arr[i] << " ";

    cout << endl;

}

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

5 6 11 12 13

**## Task 2**

- The following snippet is from `./src/insertion.cpp` lines 12-22. Discuss in groups how the code works:

```cpp

    for (int k = 1; k < 10; k++)

    {

        int temp = myarray[k];

        int j = k - 1;

        while (j >= 0 && temp <= myarray[j])

        {

            myarray[j + 1] = myarray[j];

            j = j - 1;

        }

        myarray[j + 1] = temp;

    }

```

#include <iostream>

using namespace std;

int main()

{

    int myarray[10] = {12, 4, 3, 1, 15, 45, 33, 21, 10, 2};

    cout << "\nInput list is \n";

    for (int i = 0; i < 10; i++)

    {

        cout << myarray[i] << "\t";

    }

    for (int k = 1; k < 10; k++) // k=3

    {

        int temp = myarray[k];               // temp=1

        int j = k - 1;                       // j=2

        while (j >= 0 && temp <= myarray[j]) // 4

        {

            myarray[j + 1] = myarray[j];

            j = j - 1;

        }

        myarray[j + 1] = temp;

    }

    cout << "\nSorted list is \n";

    for (int i = 0; i < 10; i++)

    {

        cout << myarray[i] << "\t";

    }

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity2\src> .\insertion

Input list is

12 4 3 1 15 45 33 21 10 2

Sorted list is

1 2 3 4 10 12 15 21 33 45

**Output:**

Input list of elements …

12  4  3  1  15  45  33  21  10  2

sorted list of elements …

1  2  3  4  10  12  15  21  33  45

In both the implementations, we can see that we begin sorting from the 2nd element of the array (loop variable j = 1) and repeatedly compare the current element to all its previous elements and then sort the element to place it in its correct position if the current element is not in order with all its previous elements.

Insertion sort works the best and can be completed in fewer passes if the array is partially sorted. But as the list grows bigger, its performance decreases. Another advantage of Insertion sort is that it is a Stable sort which means it maintains the order of equal elements in the list.

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

**## Task 3**

- Discuss the complexity analysis of insertion sort. Refer to the link below:

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

**## Links**

- <https://cpp.sh/>

**Complexity Analysis Of The Insertion Sort Algorithm**

From the pseudo code and the illustration above, insertion sort is the efficient algorithm when compared to bubble sort or selection sort. Instead of using for loop and present conditions, it uses a while loop that does not perform any more extra steps when the array is sorted.

However, even if we pass the sorted array to the Insertion sort technique, it will still execute the outer for loop thereby requiring n number of steps to sort an already sorted array. This makes the best time complexity of insertion sort a linear function of N where N is the number of elements in the array.

**Thus the various complexities for Insertion sort technique are given below:**

|  |  |
| --- | --- |
| Worst case time complexity | O(n 2 ) |
| Best case time complexity | O(n) |
| Average time complexity | O(n 2 ) |
| Space complexity | O(1) |

In spite of these complexities, we can still conclude that Insertion sort is the most efficient algorithm when compared with the other sorting techniques like Bubble sort and Selection sort.

**Conclusion**

Insertion sort is the most efficient of all the three techniques discussed so far. Here, we assume that the first element is sorted and then repeatedly compare every element to all its previous elements and then place the current element in its correct position in the array.

In this tutorial, while discussing Insertion sort we have noticed that we compare the elements using an increment of 1 and also they are contiguous. This feature results in requiring more passes to get the sorted list.

In our upcoming tutorial, we will discuss “Shell sort” which is an improvement over the Selection sort.

In shell sort, we introduce a variable known as “increment” or a “gap” using which we divide the list into sublists containing non-contiguous elements that “gap” apart. Shell sort requires fewer passes when compared to Insertion sort and is also faster.

In our future tutorials, we will learn about two sorting techniques, “Quicksort” and “Mergesort” which use “Divide and conquer” strategy for sorting data lists.

ACTIVITY 3

**# Activities**

**## Task 1:**

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

  https://www.cs.usfca.edu/~galles/visualization/QueueLL.html

**## Task 2:**

- The following snippet is from `./src/queue.cpp` lines 8-10. What happens if `front`, `rear` and `temp` were not global variables?

```cpp

struct node \*front = NULL;

struct node \*rear = NULL;

struct node \*temp;

```

**## Task 3:**

- The following snippet is from `./src/queue.cpp` lines 11-28. Discuss in groups how the code works:

```cpp

void Insert(int val)

{

    if (rear == NULL)

    {

        rear = new node;

        rear->next = NULL;

        rear->data = val;

        front = rear;

    }

    else

    {

        temp = new node;

        rear->next = temp;

        temp->data = val;

        temp->next = NULL;

        rear = temp;

    }

}

```

**## Task 4: Individual, at home**

- Discuss the various operations that can be performed on a linked list. Refer to the following link:

  https://www.softwaretestinghelp.com/linked-list/

**## Links**

- https://cpp.sh/

**## Task 1:**

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

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

A picture containing diagram

Description automatically generated

FIFO

*we maintain two pointers,****front****, and****rear****. The front**points to the first item of the queue and**rear**points to the last item.*

* ***enQueue():****This operation adds a new node after the rear**and moves the rear**to the next node.*
* ***deQueue():****This operation removes the front node and moves the front**to the next node.*

**Time Complexity:**O(1), The time complexity of both operations enqueue() and dequeue() is O(1) as it only changes a few pointers in both operations  
**Auxiliary Space:**O(1), The auxiliary Space of both operations enqueue() and dequeue() is O(1) as constant extra space is required

#include <iostream>

using namespace std;

struct QNode {

    int data;

    QNode\* next;

    QNode(int d)

    {

        data = d;

        next = NULL;

    }

};

struct Queue {

    QNode \*front, \*rear;

    Queue() { front = rear = NULL; }

    void enQueue(int x)

    {

        // Create a new LL node

        QNode\* temp = new QNode(x);

        // If queue is empty, then

        // new node is front and rear both

        if (rear == NULL) {

            front = rear = temp;

            return;

        }

        // Add the new node at

        // the end of queue and change rear

        rear->next = temp;

        rear = temp;

    }

    // Function to remove

    // a key from given queue q

    void deQueue()

    {

        // If queue is empty, return NULL.

        if (front == NULL)

            return;

        // Store previous front and

        // move front one node ahead

        QNode\* temp = front;

        front = front->next;

        // If front becomes NULL, then

        // change rear also as NULL

        if (front == NULL)

            rear = NULL;

        delete (temp);

    }

};

// Driver code

int main()

{

    Queue q;

    q.enQueue(10);

    q.enQueue(20);

    q.deQueue();

    q.deQueue();

    q.enQueue(30);

    q.enQueue(40);

    q.enQueue(50);

    q.deQueue();

    cout << "Queue Front : " << ((q.front != NULL) ? (q.front)->data : -1)<< endl;

    cout << "Queue Rear : " << ((q.rear != NULL) ? (q.rear)->data : -1);

}

<https://www.geeksforgeeks.org/queue-linked-list-implementation/>

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\queue2\_using\_linked\_list

Queue Front : 40

Queue Rear : 50

**## Task 2:**

- The following snippet is from `./src/queue.cpp` lines 8-10. What happens if `front`, `rear` and `temp` were not global variables?

```cpp

struct node \*front = NULL;

struct node \*rear = NULL;

struct node \*temp;

```

#include <iostream>

using namespace std;

struct node

{

    int data;

    struct node \*next;

};

struct node \*head = NULL;

struct node \*tail = NULL;

struct node \*temp;

void Insert(int val)

{

    if (rear == NULL)

    {

        rear = new node;

        rear->next = NULL;

        rear->data = val;

        front = rear;

    }

    else

    {

        temp = new node;

        temp->data = val;

        rear->next = temp;

        temp->next = NULL;

        rear = temp;

    }

}

void Delete()

{

    temp = front;

    if (front == NULL)

    {

        cout << "Queue is empty!!" << endl;

    }

    else if (temp->next != NULL)

    {

        temp = temp->next;

        cout << "Element deleted from queue is : " << front->data << endl;

        free(front);

        front = temp;

    }

    else

    {

        cout << "Element deleted from queue is : " << front->data << endl;

        free(front);

        front = NULL;

        rear = NULL;

    }

}

void Display()

{

    temp = front;

    if ((front == NULL) && (rear == NULL))

    {

        cout << "Queue is empty" << endl;

        return;

    }

    while (temp != NULL)

    {

        cout << temp->data << " ";

        temp = temp->next;

    }

    cout << endl;

}

int main()

{

    cout << "Queue Created:" << endl;

    Insert(10);

    Insert(20);

    Insert(30);

    Insert(40);

    Insert(50);

    Display();

    Delete();

    cout << "Queue after one deletion: " << endl;

    Display();

    return 0;

}

Starting build...

C:\mingw64\bin\g++.exe -fdiagnostics-color=always -g C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp -o C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.exe

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp: In function 'void Insert(int)':

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp:15:9: error: 'rear' was not declared in this scope

15 | if (rear == NULL)

| ^~~~

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp:20:9: error: 'front' was not declared in this scope

20 | front = rear;

| ^~~~~

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp: In function 'void Delete()':

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp:35:12: error: 'front' was not declared in this scope

35 | temp = front;

| ^~~~~

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp:52:9: error: 'rear' was not declared in this scope

52 | rear = NULL;

| ^~~~

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp: In function 'void Display()':

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp:57:12: error: 'front' was not declared in this scope

57 | temp = front;

| ^~~~~

C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture3-main\activity3\src\queue.cpp:58:29: error: 'rear' was not declared in this scope

58 | if ((front == NULL) && (rear == NULL))

| ^~~~

Build finished with error(s).

\* The terminal process terminated with exit code: -1.

YOU MUST CHANGE THE CODE

//struct node \*head = NULL;

//struct node \*tail = NULL;

struct node\* front = NULL;

struct node\* rear = NULL;

**## Task 3:**

- The following snippet is from `./src/queue.cpp` lines 11-28. Discuss in groups how the code works:

```cpp

void Insert(int val)

{

    if (rear == NULL)

    {

        rear = new node;

        rear->next = NULL;

        rear->data = val;

        front = rear;

    }

    else

    {

        temp = new node;

        rear->next = temp;

        temp->data = val;

        temp->next = NULL;

        rear = temp;

    }

}

```

**## Task 4: Individual, at home**

- Discuss the various operations that can be performed on a linked list. Refer to the following link:

<https://www.softwaretestinghelp.com/linked-list/>

**## Links**

- https://cpp.sh/

**Basic Operations on Linked List**

* Traversal : To traverse all the nodes one after another.
* Insertion : To add a node at the given position.
* Deletion : To delete a node.
* Searching : To search an element(s) by value.
* Updating : To update a node.
* Sorting: To arrange nodes in a linked list in a specific order.

**A Detailed Study Of Linked List In C++.**

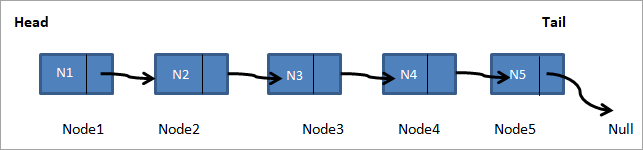
A linked list is a linear dynamic data structure to store data items. We have already seen arrays in our previous topics on basic C++. We also know that arrays are a linear data structure that store data items in contiguous locations.

Unlike arrays, the linked list does not store data items in contiguous memory locations.

A linked list consists of items called “Nodes” which contain two parts. The first part stores the actual data and the second part has a pointer that points to the next node. This structure is usually called “Singly linked list”.

We will take a look at the singly linked list in detail in this tutorial.

**The following diagram shows the structure of a singly linked list.**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/structure-of-a-singly-linked-list.png)

As shown above, the first node of the linked list is called “head” while the last node is called “Tail”. As we see, the last node of the linked list will have its next pointer as null since it will not have any memory address pointed to.

Since each node has a pointer to the next node, data items in the linked list need not be stored at contiguous locations. The nodes can be scattered in the memory. We can access the nodes anytime as each node will have an address of the next node.

We can add data items to the linked list as well as delete items from the list easily. Thus it is possible to grow or shrink the linked list dynamically. There is no upper limit on how many data items can be there in the linked list. So as long as memory is available, we can have as many data items added to the linked list.

Apart from easy insertion and deletion, the linked list also doesn’t waste memory space as we need not specify beforehand how many items we need in the linked list. The only space taken by linked list is for storing the pointer to the next node that adds a little overhead.

Next, we will discuss the various operations that can be performed on a linked list.

**We can perform various operations on a linked list as given below:**

#### **#1) Insertion**

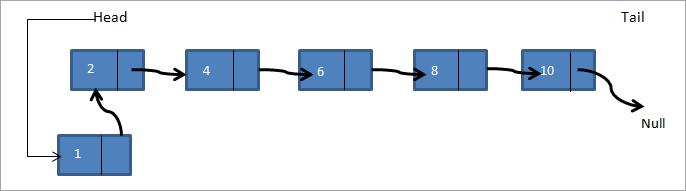
Insertion operation of linked list adds an item to the linked list. Though it may sound simple, given the structure of the linked list, we know that whenever a data item is added to the linked list, we need to change the next pointers of the previous and next nodes of the new item that we have inserted.

The second thing that we have to consider is the place where the new data item is to be added.

**There are three positions in the linked list where a data item can be added.**

**#1) At the beginning of the linked list**

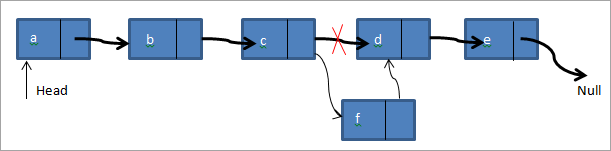
A linked list is shown below 2->4->6->8->10. If we want to add a new node 1, as the first node of the list, then the head pointing to node 2 will now point to 1 and the next pointer of node 1 will have a memory address of node 2 as shown in the below figure.

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

Thus the new linked list becomes 1->2->4->6->8->10.

**#2) After the given Node**

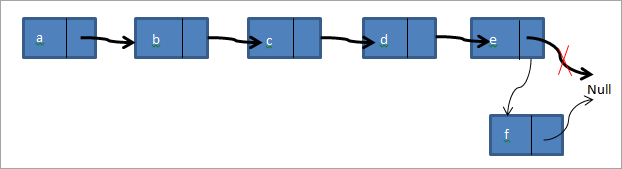
Here, a node is given and we have to add a new node after the given node. In the below-linked list a->b->c->d ->e, if we want to add a node f after node c then the linked list will look as follows:

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/to-add-a-node-f-after-node-c.png)

Thus in the above diagram, we check if the given node is present. If it’s present, we create a new node f. Then we point the next pointer of node c to point to the new node f. The next pointer of the node f now points to node d.

**#3) At the end of the Linked List**

In the third case, we add a new node at the end of the linked list. Consider we have the same linked list a->b->c->d->e and we need to add a node f to the end of the list. The linked list will look as shown below after adding the node.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/06/3.At-the-end-of-the-linked-list.png)

Thus we create a new node f. Then the tail pointer pointing to null is pointed to f and the next pointer of node f is pointed to null. We have implemented all three types of insert functions in the below C++ program.

In C++, we can declare a linked list as a structure or as a class. Declaring linked list as a structure is a traditional C-style declaration. A linked list as a class is used in modern C++, mostly while using standard template library.

In the following program, we have used structure to declare and create a linked list. It will have data and pointer to the next element as its members.

#include <iostream>

**using** **namespace** std;

// A linked list node

**struct** Node

{

**int** data;

**struct** Node \*next;

};

//insert a new node in front of the list

**void** push(**struct** Node\*\* head, **int** node\_data)

{

   /\* 1. create and allocate node \*/

**struct** Node\* newNode = **new** Node;

   /\* 2. assign data to node \*/

   newNode->data = node\_data;

   /\* 3. set next of new node as head \*/

   newNode->next = (\*head);

   /\* 4. move the head to point to the new node \*/

   (\*head) = newNode;

}

//insert new node after a given node

**void** insertAfter(**struct** Node\* prev\_node, **int** node\_data)

{

  /\*1. check if the given prev\_node is NULL \*/

**if** (prev\_node == NULL)

{

   cout<<"the given previous node is required,cannot be NULL"; **return**; }

   /\* 2. create and allocate new node \*/

**struct** Node\* newNode =**new** Node;

   /\* 3. assign data to the node \*/

   newNode->data = node\_data;

   /\* 4. Make next of new node as next of prev\_node \*/

   newNode->next = prev\_node->next;

    /\* 5. move the next of prev\_node as new\_node \*/

    prev\_node->next = newNode;

}

/\* insert new node at the end of the linked list \*/

**void** append(**struct** Node\*\* head, **int** node\_data)

{

/\* 1. create and allocate node \*/

**struct** Node\* newNode = **new** Node;

**struct** Node \*last = \*head; /\* used in step 5\*/

/\* 2. assign data to the node \*/

newNode->data = node\_data;

/\* 3. set next pointer of new node to null as its the last node\*/

newNode->next = NULL;

/\* 4. if list is empty, new node becomes first node \*/

**if** (\*head == NULL)

{

\*head = newNode;

**return**;

}

/\* 5. Else traverse till the last node \*/

**while** (last->next != NULL)

last = last->next;

/\* 6. Change the next of last node \*/

last->next = newNode;

**return**;

}

// display linked list contents

**void** displayList(**struct** Node \*node)

{

   //traverse the list to display each node

**while** (node != NULL)

   {

      cout<<node->data<<"-->";

      node = node->next;

   }

**if**(node== NULL)

cout<<"null";

}

/\* main program for linked list\*/

**int** main()

{

/\* empty list \*/

**struct** Node\* head = NULL;

// Insert 10.

append(&head, 10);

// Insert 20 at the beginning.

push(&head, 20);

// Insert 30 at the beginning.

push(&head, 30);

// Insert 40 at the end.

append(&head, 40); //

Insert 50, after 20.

insertAfter(head->next, 50);

cout<<"Final linked list: "<<endl;

displayList(head);

**return** 0;

}

**Output:**

Final linked list:

10–>20–>30–>40–>50–>null

In both the program above, C++ as well as Java, we have separate functions to add a node in front of the list, end of the list and between the lists given in a node. In the end, we print the contents of the list created using all the three methods.

#### **#2) Deletion**

Like insertion, deleting a node from a linked list also involves various positions from where the node can be deleted. We can delete the first node, last node or a random kth node from the linked list. After deletion, we need to adjust the next pointer and the other pointers in the linked list appropriately so as to keep the linked list intact.

In the following C++ implementation, we have given two methods of deletion i.e. deleting the first node in the list and deleting the last node in the list. We first create a list by adding nodes to the head. Then we display the contents of the list after insertion and each deletion.

|  |
| --- |
| #include <iostream>  **using** **namespace** std;    /\* Link list node \*/  **struct** Node {  **int** data;  **struct** Node\* next;     };    //delete first node in the linked list  Node\* deleteFirstNode(**struct** Node\* head)  {  **if** (head == NULL)  **return** NULL;       // Move the head pointer to the next node     Node\* tempNode = head;     head = head->next;  **delete** tempNode;    **return** head;  }  //delete last node from linked list  Node\* removeLastNode(**struct** Node\* head)  {  **if** (head == NULL)  **return** NULL;    **if** (head->next == NULL) {  **delete** head;  **return** NULL;     }    // first find second last node  Node\* second\_last = head;  **while** (second\_last->next->next != NULL)  second\_last = second\_last->next;    // Delete the last node  **delete** (second\_last->next);    // set next of second\_last to null  second\_last->next = NULL;    **return** head;  }    // create linked list by adding nodes at head  **void** push(**struct** Node\*\* head, **int** new\_data)  {  **struct** Node\* newNode = **new** Node;     newNode->data = new\_data;     newNode->next = (\*head);     (\*head) = newNode;  }    // main function  **int** main()  {     /\* Start with the empty list \*/     Node\* head = NULL;       // create linked list     push(&head, 2);     push(&head, 4);     push(&head, 6);     push(&head, 8);     push(&head, 10);             Node\* temp;       cout<<"Linked list created "<<endl; **for** (temp = head; temp != NULL; temp = temp->next)     cout << temp->data << "-->";  **if**(temp == NULL)     cout<<"NULL"<<endl;           //delete first node     head = deleteFirstNode(head);     cout<<"Linked list after deleting head node"<<endl; **for** (temp = head; temp != NULL; temp = temp->next)     cout << temp->data << "-->";  **if**(temp == NULL)     cout<<"NULL"<<endl;          //delete last node     head = removeLastNode(head);     cout<<"Linked list after deleting last node"<<endl; **for** (temp = head; temp != NULL; temp = temp->next)     cout << temp->data << "-->";  **if**(temp == NULL)     cout<<"NULL";    **return** 0;  } |

**Output:**

Linked list created

10–>8–>6–>4–>2–  
>NULL

Linked list after deleting head node

8–>6–>4–>2–  
>NULL

Linked list after deleting last node

8–>6–>4–>NULL

| **Arrays** | **Linked lists** |
| --- | --- |
| Arrays have fixed size | Linked list size is dynamic |
| Insertion of new element is expensive | Insertion/deletion is easier |
| Random access is allowed | Random access not possible |
| Elements are at contiguous location | Elements have non-contiguous location |
| No extra space is required for the next pointer | Extra memory space required for next pointer |

**Applications**

As arrays and linked lists are both used to store items and are linear data structures, both these structures can be used in similar ways for most of the applications.

**Some of the applications for linked lists are as follows:**

* A linked list can be used to implement stacks and queues.
* A linked list can also be used to implement graphs whenever we have to represent graphs as adjacency lists.
* A mathematical polynomial can be stored as a linked list.
* In the case of hashing technique, the buckets used in hashing are implemented using the linked lists.
* Whenever a program requires dynamic allocation of memory, we can use a linked list as linked lists work more efficiently in this case.

**Conclusion**

Linked lists are the data structures that are used to store data items in a linear fashion but noncontiguous locations. A linked list is a collection of nodes that contain a data part and a next pointer that contains the memory address of the next element in the list.

The last element in the list has its next pointer set to NULL, thereby indicating the end of the list. The first element of the list is called the Head. The linked list supports various operations like insertion, deletion, traversal, etc. In case of dynamic memory allocation, linked lists are preferred over arrays.

Linked lists are expensive as far as their traversal is concerned since we cannot randomly access the elements like arrays. However, insertion-deletion operations are less expensive when compared arrays.

We have learned all about linear linked lists in this tutorial. Linked lists can also be circular or doubly. We will have an in-depth look at these lists in our upcoming tutorials.