

INF218

Struktur Data & Algoritma

Pendahuluan

Alim Misbullah, S.Si., M.S.

Kontrak Kuliah

- Mata Kuliah : Struktur Data & Algoritma 3 SKS (2-1)
- Pengajar : Alim Misbullah, S.Si., M.S.
- Kontak : Group WA (**No Personal Chat**)
- Komting : Riska Adelia (STK)
 : Abdul Hafidh (INF)

Kontrak Kuliah

- Mata kuliah SDA akan membahas tentang konsep-konsep yang digunakan untuk mempercepat kinerja sebuah program seperti linked list, queue, stack, sorting, searching sehingga waktu yang diperlukan untuk menyelesaikan sebuah masalah akan lebih cepat dari metode sederhana lainnya
- Pada mata kuliah ini, mahasiswa diharapkan sudah memiliki kemampuan dasar programming khususnya bahasa pemrograman C sehingga nantinya akan lebih mudah dalam melakukan implementasi terhadap konsep-konsep yang diajarkan.

Review...

- Sebuah program komputer adalah sekumpulan perintah untuk menyelesaikan masalah tertentu. Dalam hal ini, program komputer perlu untuk menyimpan data, membaca data dan melakukan perhitungan pada data tersebut.
- Struktur data merupakan sebuah istilah yang digunakan untuk menyimpan dan mengorganisir data. Sedangkan, algoritma adalah kumpulan langkah-langkah untuk menyelesaikan masalah tertentu
- Struktur Data dan Algoritma merupakan salah satu konsep yang harus dipelajari agar kita dapat menulis program komputer secara efisien dan optimal

Apa itu Struktur Data?

Struktur Data

- Struktur data adalah sebuah cara yang digunakan untuk mengorganisir data sehingga data tersebut dapat digunakan secara efisien (waktu dan tempat).
- Struktur data sangat erat kaitannya dengan penggunaan memori pada komputer sehingga data yang disimpan akan dengan mudah dapat diakses dan dilakukan manipulasi seperti penambahan, pencarian, penghapusan, pengeditan
- Struktur data **bukan** bahasa pemrograman seperti C, C++, Java, dll. SD merupakan kumpulan algoritma yang dapat diimplementasikan dalam berbagai bahasa pemrograman

Jenis Struktur Data

- Primitive Data Structure
 - Primitive Struktur Data adalah tipe data primitive seperti **int**, **float**, **double**, **char** dan **pointer** yang hanya dapat menampung satu nilai saja
 - `int a = 50; // variable a bertipe integer (-2^32 s.d 2^32-1), 4 bytes = 32 bits`
- Non-primitive Data Structure
 - Linear Data Structure
 - Data yang tersusun secara sequential (berurutan) dimana satu elemen terkoneksi hanya dengan satu element lainnya, seperti **array**, **linked list**, **stacks**, dan **queue**
 - `int arr[5] = {1}`
 - Non-linear Data Structure
 - Elemen yang terkoneksi ke banyak elemen lainnya, seperti **tree** dan **graph**

Jenis Struktur Data

- Static Data Structure
 - Jenis struktur data dimana ukuran memori akan dialokasikan pada saat kode atau program dikompilasi. Oleh karena itu, ukuran maksimum sudah pasti.
- Dynamic Data Structure
 - Jenis struktur data dimana ukuran memori akan dialokasikan pada saat kode atau program dijalankan. Oleh karena itu, ukuran maksimumnya sangat fleksibel

Operasi Pada Struktur Data

- Searching (Pencarian)
- Sorting (Pengurutan)
- Insertion (Penambahan)
- Updation (Pengeditan)
- Deletion (Penghapusan)

Keuntungan Struktur Data

- Efficiency (Efisiensi)
- Reusability (Digunakan berulang-ulang)
- Abstraction (Bias)

Apa itu Algoritma?

Algoritma

- Algoritma adalah sekumpulan instruksi (perintah) atau langkah-langkah yang tersusun dengan rapi untuk menyelesaikan sebuah masalah
- Kualitas sebuah Algoritma:
 - Input dan output dapat didefiniskan dengan tepat
 - Setiap langkah dalam sebuah algoritma harus jelas dan tidak ambigu
 - Algoritma yang digunakan adalah yang paling efektif diantara banyak solusi untuk menyelesaikan masalah
 - Sebuah algoritma seharusnya tidak memasukkan kode program. Bahkan, algoritma seharusnya ditulis untuk diimplementasikan dalam berbagai bahasa pemrograman

Contoh Algoritma

Algorithm to add two numbers entered by the user

```
Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
        sum←num1+num2
Step 5: Display sum
Step 6: Stop
```

<https://www.programiz.com/dsa/algorithm#add>

Contoh Algoritma

Find the largest number among three different numbers

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: Read variables a,b and c.
Step 4: If a > b
    If a > c
        Display a is the largest number.
    Else
        Display c is the largest number.
    Else
        If b > c
            Display b is the largest number.
        Else
            Display c is the greatest number.
Step 5: Stop
```

Contoh Algoritma Lainnya...

Algorithm to find all the roots of the quadratic equation

Algorithm to find the factorial

Algorithm to check prime number

Algorithm of Fibonacci series

Mengapa Perlu Belajar Algoritma?

Mengapa Perlu Algoritma?

- Waktu sangat berharga
 - Penyelesaian sebuah masalah (problem) di komputer akan lebih singkat jika memiliki algoritma yang efisien.
 - Contoh: Hitunglah penjumlahan deretan bilangan 10^{11}
 - Algoritma Sederhana

```
Initialize sum = 0
for every natural number n in range 1 to 1011 (inclusive):
    add n to sum
sum is your answer
```

- Code

```
int findSum() {
    int sum = 0;
    for (int v = 1; v <= 100000000000; v++) {
        sum += v;
    }
    return sum;
}
```

Butuh waktu setidaknya 16 menit

Mengapa Perlu Algoritma?

- Algoritma yang efisien

```
Sum = N * (N + 1) / 2
```

- Code

```
int sum(int N) {  
    return N * (N + 1) / 2;  
}
```

Hanya butuh milisecond

Mengapa Perlu Algoritma?

- Skalabilitas
 - Artinya, sebuah algoritma akan mampu bekerja dengan baik pada space yang lebih besar atau ukuran data yang lebih besar
 - Solusi pertama pada contoh sebelumnya disebut dengan **Linearly Scalable Algorithm** dan solusi kedua disebut dengan **Constant-Time Algorithm**
- Memori is Expensive
 - Contoh sederhana, kita dapat menyimpan informasi **umur (age)** dari sebuah entity tanpa perlu menyimpan tanggal lahir karena kita dapat menghitung tanggal lahir pada saat run time

Algoritma menjadi konsep dasar untuk melakukan optimisasi terhadap kode program sehingga menghemat waktu dan memori

Referensi

- <https://www.programiz.com/dsa>
- <https://www.javatpoint.com/data-structure-tutorial>
- [https://www.tutorialspoint.com/data structures algorithms/index.htm](https://www.tutorialspoint.com/data_structures_algorithms/index.htm)

INF218

Struktur Data & Algoritma

Linked List

Alim Misbullah, S.Si., M.S.

Review Array...

- Array didefinisikan sebagai kumpulan data yang memiliki tipe data yang sama yang saling berdekatan di dalam memori
- Array diturunkan dari tipe data primitive di dalam bahasa C seperti int, float, char, double, dll
- Elemen dari array dapat diakses menggunakan index dari elemen tersebut
- Setiap element memiliki tipe data dan ukuran yang sama, seperti int = 4 bytes ($\text{int } a[10] = 10 * 4 = 40 \rightarrow a[0]$ untuk elemen 1)
- Array sangat diperlukan untuk data yang berbeda-beda namun memiliki tipe yang sama, seperti nilai mahasiswa

Review Array...

Time Complexity

Algorithm	Average Case	Worst Case
Access	$O(1)$	$O(1)$
Search	$O(n)$	$O(n)$
Insertion	$O(n)$	$O(n)$
Deletion	$O(n)$	$O(n)$

Space Complexity

Pada array, space complexity adalah **$O(n)$**

*Watch this video to understand more about **O** notation: https://www.youtube.com/watch?v=_vX2sjlpXU*

Review Struct dan Pointer...

```
#include <stdio.h>
struct person
{
    int age;
    float weight;
};

int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);

    return 0;
}
```

Sebelum ke Linked List

int x = 6; → 4 bytes in memory

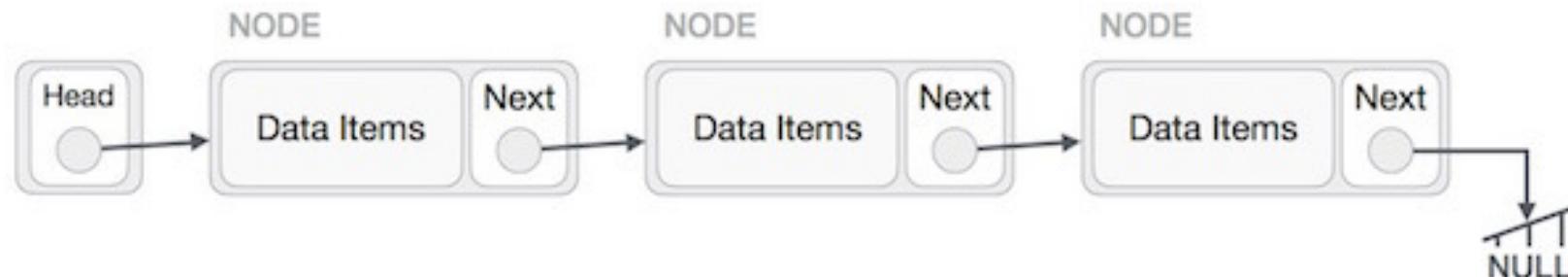
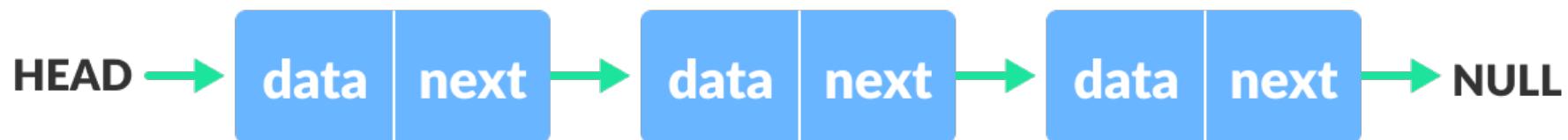
int x[**10**] = {5, 7, 9}; → ? ($4 \times 3 = 12$)

Kekurangan array:

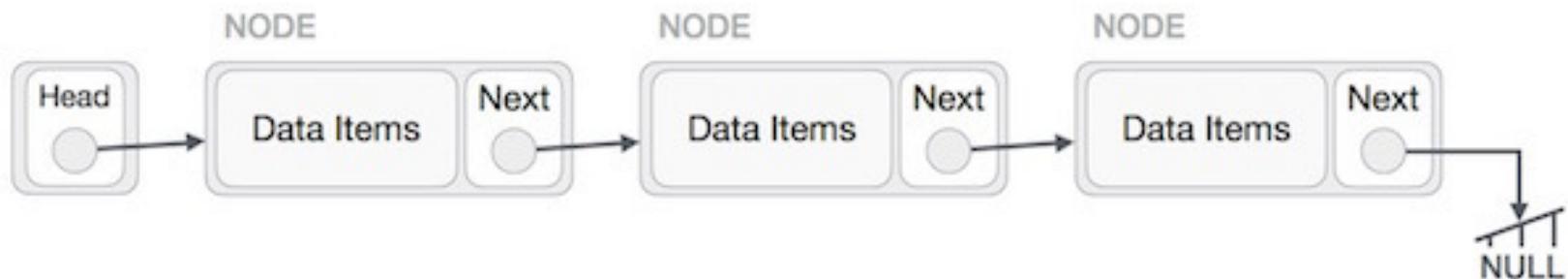
- Tidak dapat menambahkan elemen lebih dari yang sudah di deklarasikan
- Pada array, banyak sekali memory yang akan terbuang dan memory ini tidak dapat digunakan oleh variabel yang lain karena sudah dialokasikan untuk array tersebut

Definisi Linked List

- Linked List merupakan kumpulan **node** yang saling terhubung di dalam memori. Node tersebut terdiri dari 2 bagian yaitu **data** dan **alamat** dari berikutnya



Deklarasi Linked List



- **Head** adalah permulaan dari sebuah Linked List
- Setiap node merepresentasikan **data** dan **alamat** dari data berikutnya

```
struct node
{
    int data;
    struct node *next;
};
```

Deklarasi Linked List

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

```
struct node
{
    int data;
    struct node *next;
};
```

Deklarasi Linked List



- Linked memiliki kelebihan untuk memutuskan rantai dan menggabungkan kembali tanpa merusak struktur element. Misalnya, kita ingin menambahkan nilai 4 diantara nilai 1 dan 2, caranya:
 - Buat struct node yang baru dan alokasi memori ke node tersebut
 - Tambahkan nilai 4
 - Gunakan pointer dari node baru ke nilai 2
 - Ubah pointer dari node yang nilai 1 ke nilai 4 yang baru ditambahkan

Demo Simple Linked List

<https://www.programiz.com/dsa/linked-list#c-code>

Operasi Pada Linked List

Traverse yaitu menampilkan elemen linked list

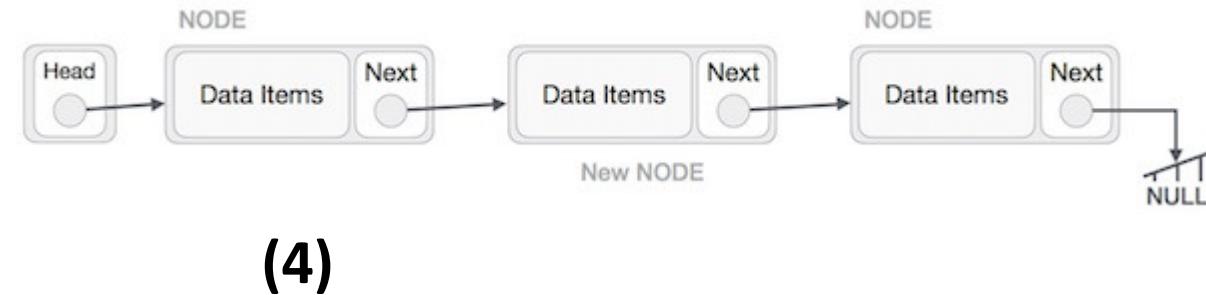
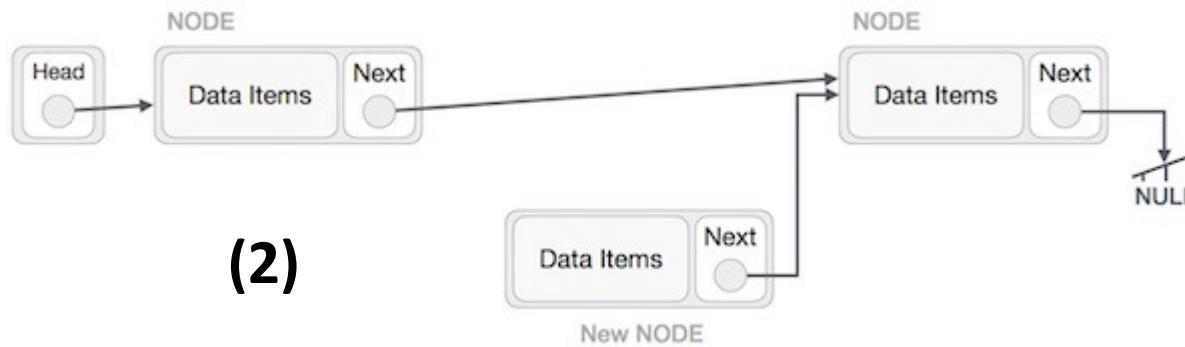
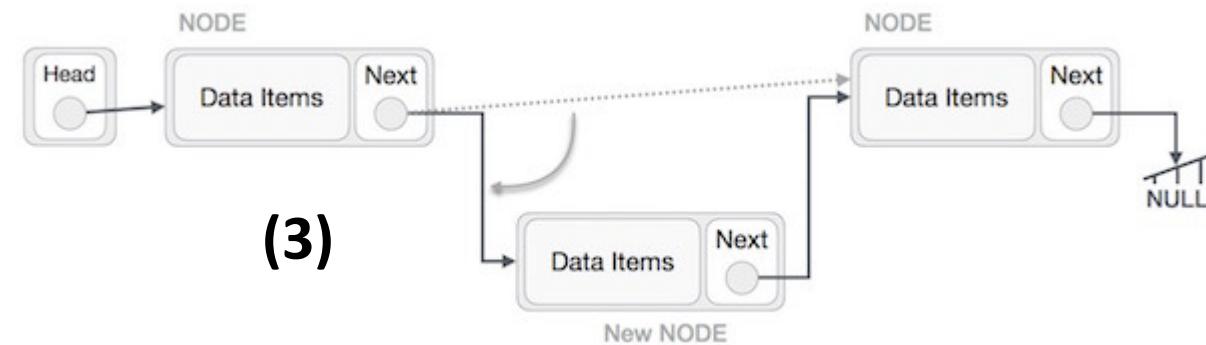
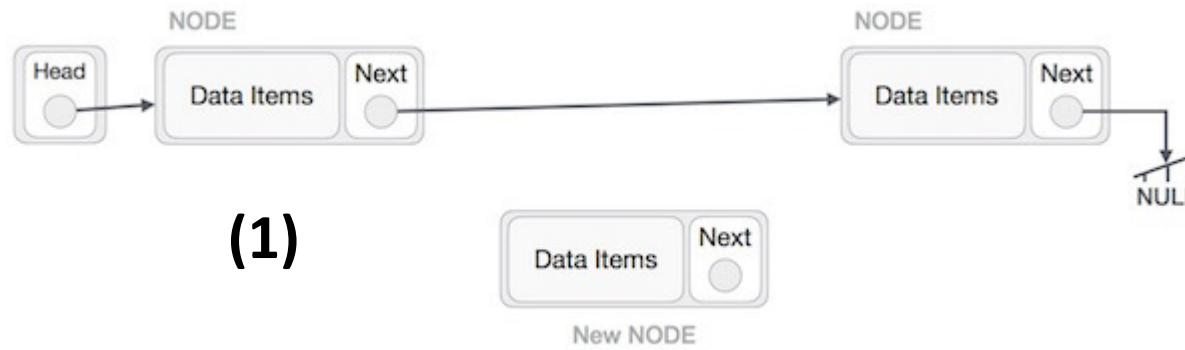
```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
    printf("%d --->", temp->data);
    temp = temp->next;
}
```

Outputnya adalah:

```
List elements are -
1 --->2 --->3 --->
```

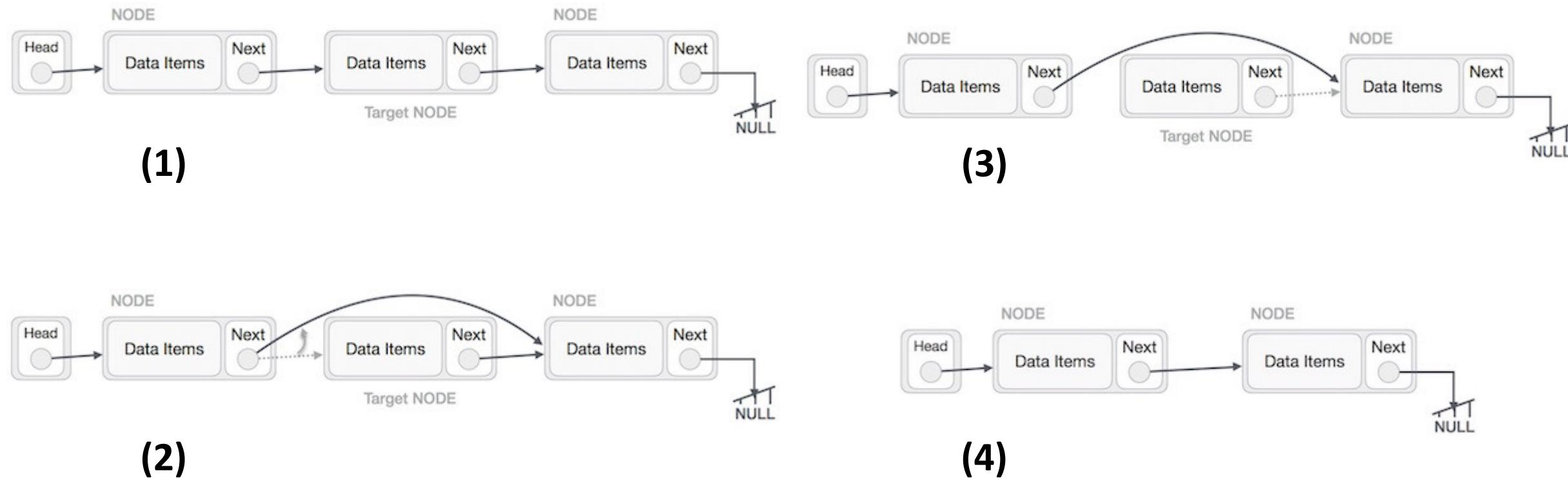
Operasi Pada Linked List

Insertion yaitu menambahkan elemen ke linked list



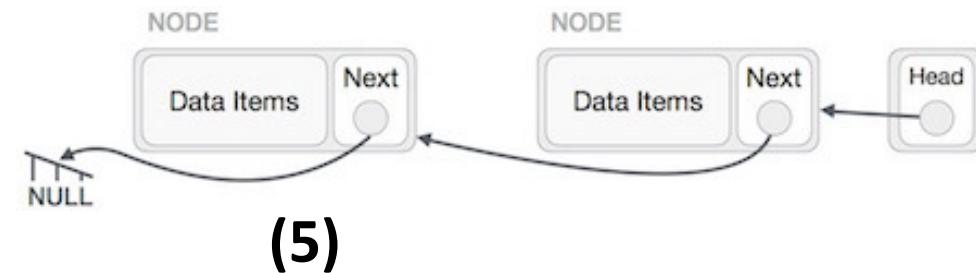
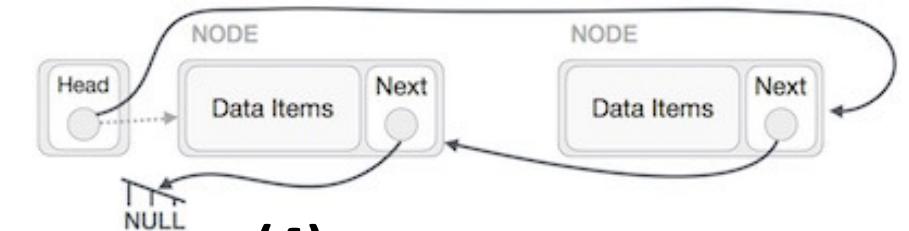
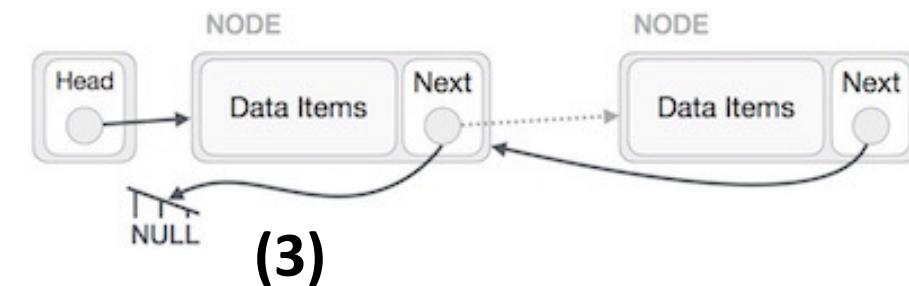
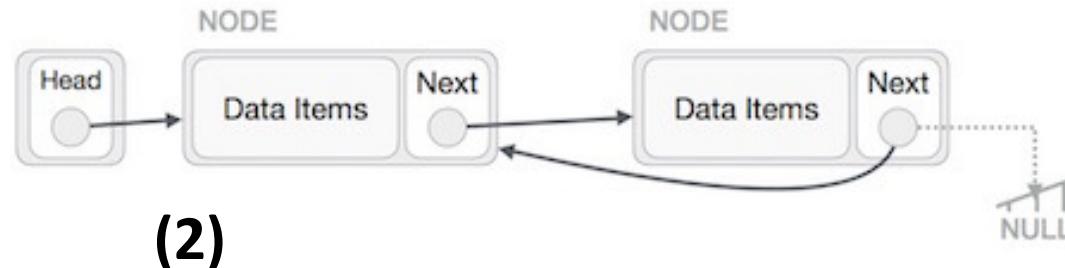
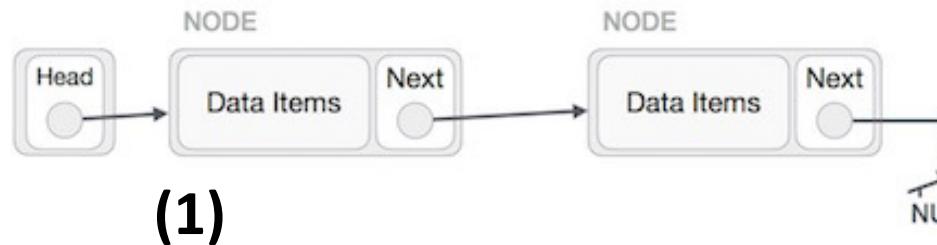
Operasi Pada Linked List

Deletion yaitu menghapus elemen dari linked list



Operasi Pada Linked List

Reverse yaitu membalikkan posisi elemen dari linked list



Demo Operasi Pada Linked List

https://www.tutorialspoint.com/data_structures_algorithms/linked_list_program_in_c.htm

Kompleksitas Linked List

Time Complexity

	Worst case	Average Case
Search	$O(n)$	$O(n)$
Insert	$O(1)$	$O(1)$
Deletion	$O(1)$	$O(1)$

Space Complexity: $O(n)$

Tipe Linked List

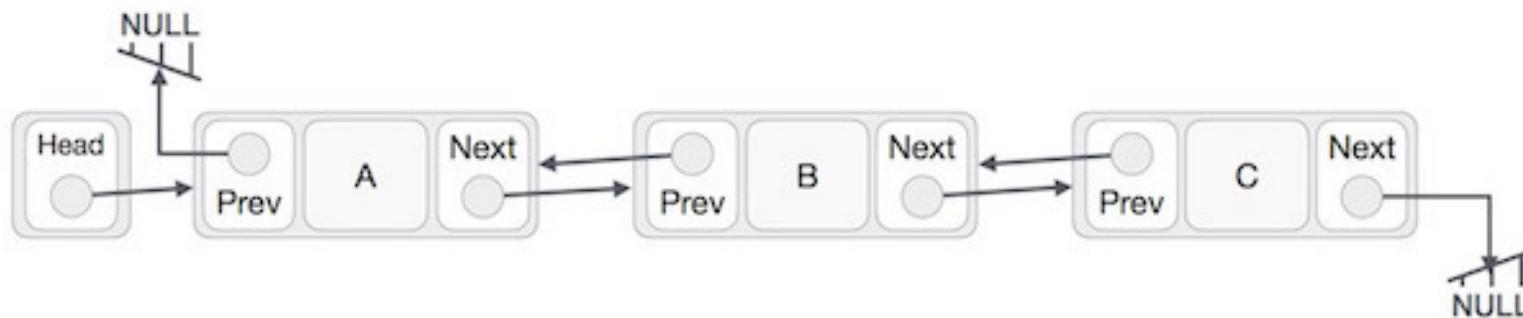
- Singly Linked List
 - Setiap node memiliki data/nilai dan pointer ke node berikutnya
- Doubly Linked List
 - Setiap node memiliki pointer ke node sebelumnya dan berikutnya sehingga linked list memiliki 2 arah (forward dan backward)
- Circular Linked List
 - Akhir dari sebuah node selalu terhubung ke node yang pertama

Tipe Linked List: Singly Linked List



```
struct node {  
    int data;  
    struct node *next;  
}
```

Tipe Linked List: Doubly Linked List



```
struct node {  
    int data;  
    struct node *next;  
    struct node *prev;  
}
```

Beberapa poin penting dari Doubly Linked List:

- Doubly Linked List mengandung sebuah link elemen yang disebut dengan **first** dan **last**
- Setiap link membawa sebuah data dan dua link yang disebut dengan **next** and **previous**
- Setiap link terhubung dengan link berikutnya menggunakan **next** link
- Setiap link terhubung dengan link sebelumnya menggunakan **previous** link
- Link terakhir ditandai dengan **null** yang menandakan akhir dari sebuah list

Tipe Linked List: Doubly Linked List

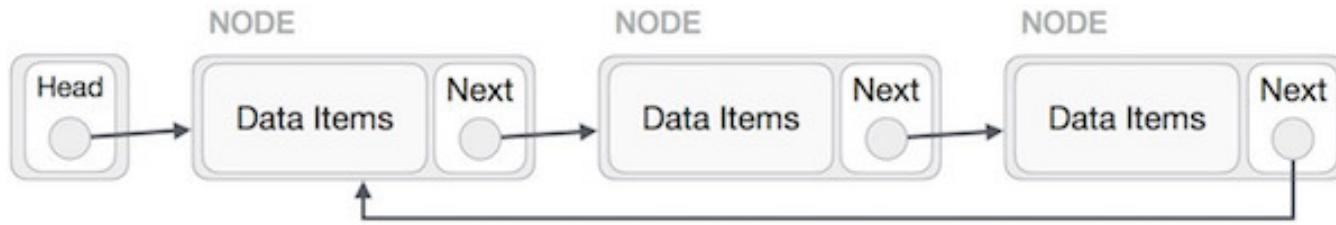
- Operasi dasar pada Doubly Linked List:
 - Insertion: Menambahkan sebuah elemen pada bagian awal list
 - Deletion: Menghapus sebuah element pada bagian awal list
 - Insert Last: Menambahkan sebuah element pada bagian akhir list
 - Delete Last: Menghapus sebuah elemen pada bagian akhir list
 - Delete: Menghapus sebuah elemen dari list menggunakan key
 - Display forward: Menampilkan semua list dari awal ke akhir
 - Display backward: Menampilkan semua list dari akhir ke awal

Demo Doubly Linked List

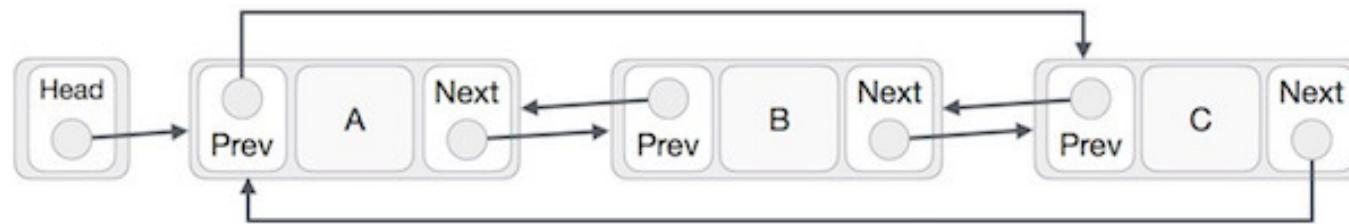
https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_program_in_c.htm

Tipe Linked List: Circular Linked List

Singly Linked List as Circular



Doubly Linked List as Circular



Beberapa poin penting pada Circular Linked List:

- Link yang ada pada list terakhir akan mengarah ke link pertama pada kedua jenis linked list
- Link **prev** pada node pertama akan mengarah ke link **prev** pada node terakhir

```
/* Connect nodes */  
one->next = two;  
two->next = three;  
three->next = one;
```

Tipe Linked List: Circular Linked List

- Operasi dasar pada Circular Linked List:
 - Insert: Menambahkan sebuah elemen pada bagian awal dari list
 - Delete: Menghapus sebuah elemen pada bagian awal dari list
 - Display: Menampilkan list

Demo Circular Linked List

https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_program_in_c.htm

Aplikasi Linked List

- Dynamic Memory Allocation
- Diimplementasikan pada Stack dan Queue
- Diaplikasi pada fitur **undo** dari sebuah software
- Hash table dan graphs

Contoh Linked List untuk Praktikum Mandiri

- Java
 - <https://www.programiz.com/java-programming/examples/get-middle-element-of-linkedlist>
 - <https://www.programiz.com/java-programming/examples/linkedlist-array-conversion>
 - <https://www.programiz.com/java-programming/examples/detect-loop-in-linkedlist>
- C
 - https://www.tutorialspoint.com/data_structures_algorithms/linked_list_program_in_c.htm
 - https://www.tutorialspoint.com/data_structures_algorithms/doubly_linked_list_program_in_c.htm
 - https://www.tutorialspoint.com/data_structures_algorithms/circular_linked_list_program_in_c.htm

Referensi

- <https://www.programiz.com/dsa/linked-list>
- <https://www.javatpoint.com/ds-linked-list>
- https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm
- <https://www.geeksforgeeks.org/data-structures/linked-list/>

INF218

Struktur Data & Algoritma

Stack

Alim Misbullah, S.Si., M.S.

Definisi Stack

Definisi Stack

- Stack merupakan sebuah Abstract Data Type (ADT) yang digunakan pada bahasa pemrograman
- Dinamai dengan **Stack** karena sesuai dengan aplikasi di dunia nyata seperti tumpukan kartu, piring, dan lain-lain



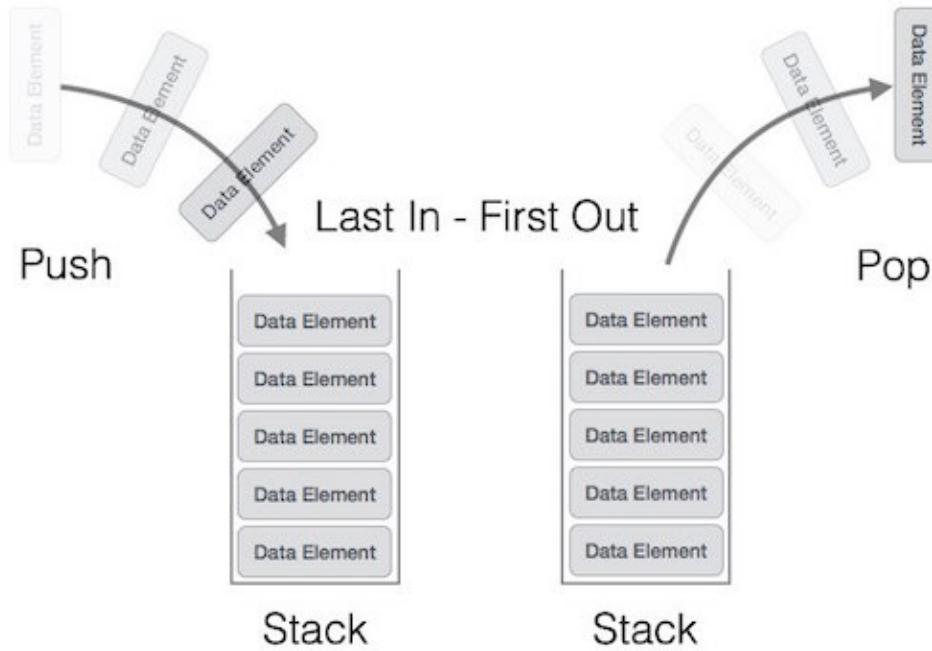
https://www.tutorialspoint.com/data_structures_algorithms/images/stack_example.jpg

- Pada kehidupan nyata, operasi stack hanya berlaku pada satu sisi saja yaitu bagian atas dari sebuah stack, seperti mengambil kartu atau piring dari tumpukan

Definisi Stack

- Sama halnya dengan Stack ADT, operasi yang berlaku juga hanya pada satu sisi saja yaitu bagian atas dari sebuah tumpukan sehingga elemen yang dapat diakses hanya bagian atasnya saja
- Stack disebut sebagai **LIFO (Last In First Out)** data structure yang berarti bahwa elemen yang ditambahkan terakhir akan diakses pertama
- Operasi penambahan elemen disebut dengan **PUSH** dan operasi penghapusan elemen disebut dengan **POP**

Definisi Stack



https://www.tutorialspoint.com/data_structures_algorithms/images/stack_representation.jpg

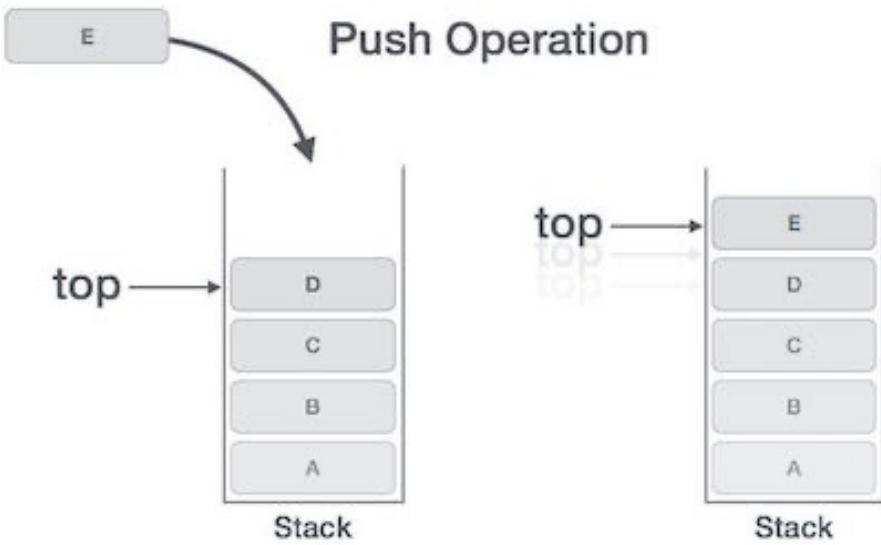
- Stack dapat diimplementasikan menggunakan Array, Struct, Pointer dan Linked List.
- Stack dapat memiliki ukuran yang tetap atau ukuran yang dinamis

Operasi Pada Stack

Operasi PUSH

- Operasi **Push** → Proses untuk menambahkan data baru ke dalam stack. Langkah-langkahnya:
 - Langkah 1: Cek apakah stack sudah penuh (`isFull`)
 - Langkah 2: Jika stack sudah penuh, tampilkan error/warning dan keluar dari program
 - Langkah 3: Jika stack tidak penuh, arahkan pointer **top** ke bagian kosong dari stack
 - Langkah 4: Tambahkan data elemen ke lokasi stack yang ditunjuk oleh pointer **top**
 - Langkah 5: Return sukses

Operasi PUSH



Algoritma

```
begin procedure push: stack, data  
  
    if stack is full  
        return null  
    endif  
  
    top ← top + 1  
    stack[top] ← data  
  
end procedure
```

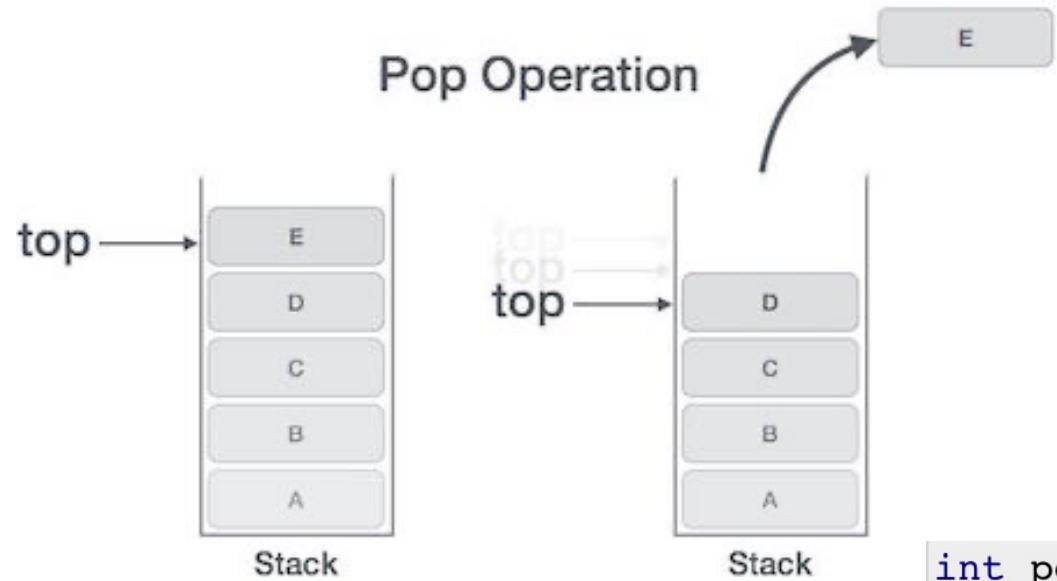
Implementasi

```
void push(int data) {  
    if(!isFull()) {  
        top = top + 1;  
        stack[top] = data;  
    } else {  
        printf("Could not insert data, Stack is full.\n");  
    }  
}
```

Operasi POP

- Operasi **POP** → Proses untuk menghapus sebuah elemen di dalam stack.
- Pada implementasi **POP** dengan array, data elemen tidak dihapus secara permanen, namun **top** diturunkan ke posisi dibawahnya untuk menunjuk ke nilai selanjutnya.
- Sedangkan pada linkedlist, **POP** akan menghapus data elemen dan membatalkan alokasi memori
- Langkah-Langkah pada operasi **POP** adalah:
 - Langkah 1: Mengecek apakah stack kosong
 - Langkah 2: Jika stack kosong, tampilkan error/warning dan keluar dari program
 - Langkah 3: Jika stack tidak kosong, akses data elemen yang ditunjuk oleh **top**
 - Langkah 4: Kurangi nilai **top** sebanyak 1
 - Langkah 5: Return sukses

Operasi POP



Algoritma

```
begin procedure pop: stack  
  
    if stack is empty  
        return null  
    endif  
  
    data ← stack[top]  
    top ← top - 1  
    return data  
  
end procedure
```

```
int pop(int data) {  
  
    if(!isempty()) {  
        data = stack[top];  
        top = top - 1;  
        return data;  
    } else {  
        printf("Could not retrieve data, Stack is empty.\n");  
    }  
}
```

Fungsi pada Stack

- Untuk menggunakan stack secara efisien, kita juga perlu mengecek status dari sebuah stack. Beberapa fungsi yang digunakan untuk mengecek status sebuah stack adalah sebagai berikut:
 - peek() → mendapatkan elemen data teratas dari stack, tanpa menghapusnya
 - isFull() → mengecek apakah stack penuh
 - isEmpty() → mengecek apakah stack kosong
- Pada stack, sebuah pointer selalu akan menunjuk ke elemen data terakhir yang dimasukkan ke dalam stack, sehingga pointer tersebut dinamai dengan **top**

Fungsi pada Stack

Algoritma dari peek()

```
begin procedure peek
    return stack[top]
end procedure
```

Algoritma dari isEmpty()

```
begin procedure isempty
    if top equals to MAXSIZE
        return true
    else
        return false
    endif
end procedure
```

Implementasi dari peek()

```
int peek() {
    return stack[top];
}
```

Implementasi dari isEmpty()

```
bool isfull() {
    if (top == MAXSIZE)
        return true;
    else
        return false;
}
```

Algoritma dari isEmpty()

```
begin procedure isempty
    if top less than 1
        return true
    else
        return false
    endif
end procedure
```

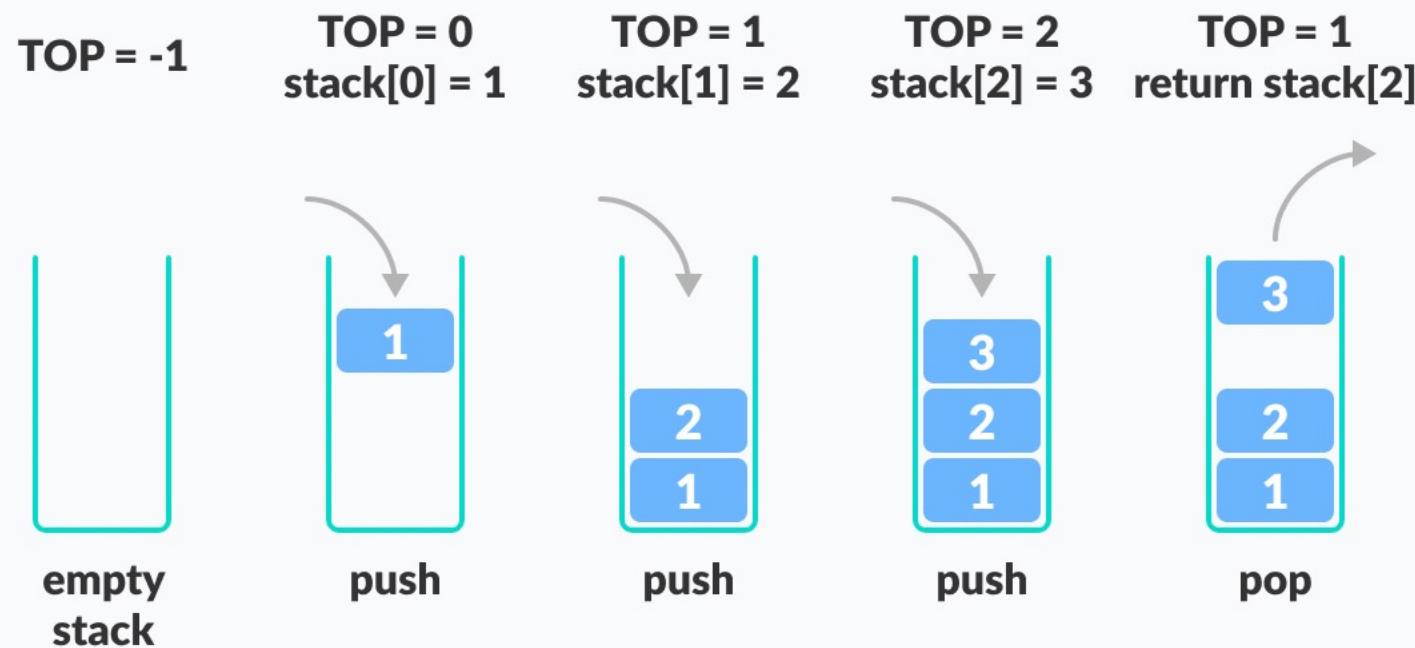
Implementasi dari isEmpty()

```
bool isempty() {
    if (top == -1)
        return true;
    else
        return false;
}
```

Proses Kerja Stack

1. Sebuah pointer disebut sebagai **TOP** digunakan untuk selalu menunjuk data elemen pada bagian atas stack
2. Ketika menginisialkan sebuah stack, kita memberikan nilai -1 untuk **TOP** sehingga kita dapat mengecek jika stack tersebut kosong dengan membandingkan nilai **TOP == -1**
3. Pada saat memasukkan sebuah data elemen ke stack, kita menaikkan nilai **TOP** dan tempatkan data elemen tersebut pada posisi yang ditunjuk oleh **TOP**
4. Pada saat mengeluarkan sebuah data elemen dari stack, kita mengembalikan data elemen yang ditunjuk oleh **TOP** dan menurunkan nilai dari **TOP** tersebut
5. Sebelum memasukkan data elemen ke stack, kita mengecek jika stack sudah penuh atau tidak
6. Sebelum mengeluarkan data elemen dari stack, kita mengecek jika stack sudah kosong

Proses Kerja Stack



Working of Stack Data Structure

Demo Implementasi Stack

- Bahasa C
 - https://www.tutorialspoint.com/data_structures_algorithms/stack_program_in_c.htm
 - <https://www.programiz.com/dsa/stack#c-code>
- Java
 - <https://www.programiz.com/dsa/stack#java-code>
- Python
 - <https://www.programiz.com/dsa/stack#python-code>

Implementasi Stack

- Menggunakan Array
 - <https://www.javatpoint.com/ds-array-implementation-of-stack>
- Menggunakan Linked List
 - <https://www.javatpoint.com/ds-linked-list-implementation-of-stack>

Expression Parsing

Definisi Parsing Ekspresi

- Cara untuk menulis sebuah ekspresi aritmatika dikenal dengan istilah **notasi**. Sebuah expresi aritmatika dapat ditulis dalam 3 cara untuk ekspresi aritmatika yang sama tanpa mengubah hasil operasi dari aritmatika tersebut, yaitu:
 - Infix Notation
 - Prefix (Polish) Notation
 - Postfix (Reverse-Polish) Notation

Definisi Parsing Ekspresi

- **Infix Notation** → Operator diletakkan diantara operand. Ini akan memudahkan manusia dalam membaca, menulis dan menghitung ekspresi aritmatika tersebut. Namun, notasi ini akan sulit ketika diimplementasikan pada komputer karena akan menghabiskan banyak waktu dan memory.
 - Contoh infix notation: $a + b + c$
- **Prefix Notation** → Operator diletakkan tepat sebelum operand. Prefix notation juga dikenal dengan nama **Polish Notation**
 - Contoh prefix (polish) notation: $+ab$ → infix notationnya $a + b$

Definisi Parsing Ekspresi

- **Postfix Notation** → Operator diletakkan tepat setelah operand. Notasi ini juga dikenal dengan istilah **Reversed Polish Notation**
 - Contoh postfix (reverse-polish) notation: $ab+$ → infix notation $a + b$

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	$a + b$	$+ a b$	$a b +$
2	$(a + b) * c$	$* + a b c$	$a b + c *$
3	$a * (b + c)$	$* a + b c$	$a b c + *$
4	$a / b + c / d$	$+ / a b / c d$	$a b / c d / +$
5	$(a + b) * (c + d)$	$* + a b + c d$	$a b + c d + *$
6	$((a + b) * c) - d$	$- * + a b c d$	$a b + c * d -$

Operasi Parsing Expression

- **Precedence (Hak Lebih Tinggi)** → Ketika sebuah operand berada diantara dua operator berbeda, maka operator yang melakukan operasi lebih dulu adalah yang memiliki hak lebih tinggi
 - Contoh: $a + b * c \rightarrow a + (b * c) \rightarrow \text{Prefix Notation: } *c+ab$
- **Associativity (Asosiatif)** → Jika dua operator memiliki hak yang sama, maka operasinya ditentukan oleh sifat asosiatif
 - Contoh: $a + b - c \rightarrow (a + b) - c$

Operasi Parsing Expression

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (-)	Lowest	Left Associative

Implementasi Parsing Expression

- Postfix Evaluation Algorithm

```
Step 1 - scan the expression from left to right
```

```
Step 2 - if it is an operand push it to stack
```

```
Step 3 - if it is an operator pull operand from stack and perform operation
```

```
Step 4 - store the output of step 3, back to stack
```

```
Step 5 - scan the expression until all operands are consumed
```

```
Step 6 - pop the stack and perform operation
```

- Contoh C code untuk Postfix Evaluation

- https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing_using_stack.htm

Q & A

Referensi

- [https://www.tutorialspoint.com/data structures algorithms/stack algorithm.htm](https://www.tutorialspoint.com/data_structures_algorithms/stack_algorithm.htm)
- <https://www.javatpoint.com/data-structure-stack>
- <https://www.programiz.com/dsa/stack>
- [https://www.tutorialspoint.com/data structures algorithms/expression parsing.htm](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm)

INF218

Struktur Data & Algoritma

Queue

Alim Misbullah, S.Si., M.S.

Definisi Queue

Definisi Queue

- **Queue** juga merupakan sebuah Abstract Data Type (ADT) yang digunakan pada bahasa pemrograman, seperti halnya stack
- Dinamai dengan **Queue** karena sesuai dengan aplikasi di dunia nyata seperti antrian pada pembelian tiket, bus stop di halte, dan lain-lain



- Pada kehidupan nyata, operasi queue berlaku pada kedua sisi yaitu satu bagian untuk menambahkan data (**enqueue**) dan satu bagian lainnya untuk mengeluarkan data (**dequeue**)

Definisi Queue

- Queue disebut sebagai **FIFO (First In First Out)** data structure yang berarti bahwa elemen yang ditambahkan pertama akan diakses pertama juga



- Operasi penambahan elemen disebut dengan **enqueue()** dan operasi penghapusan elemen disebut dengan **dequeue()**

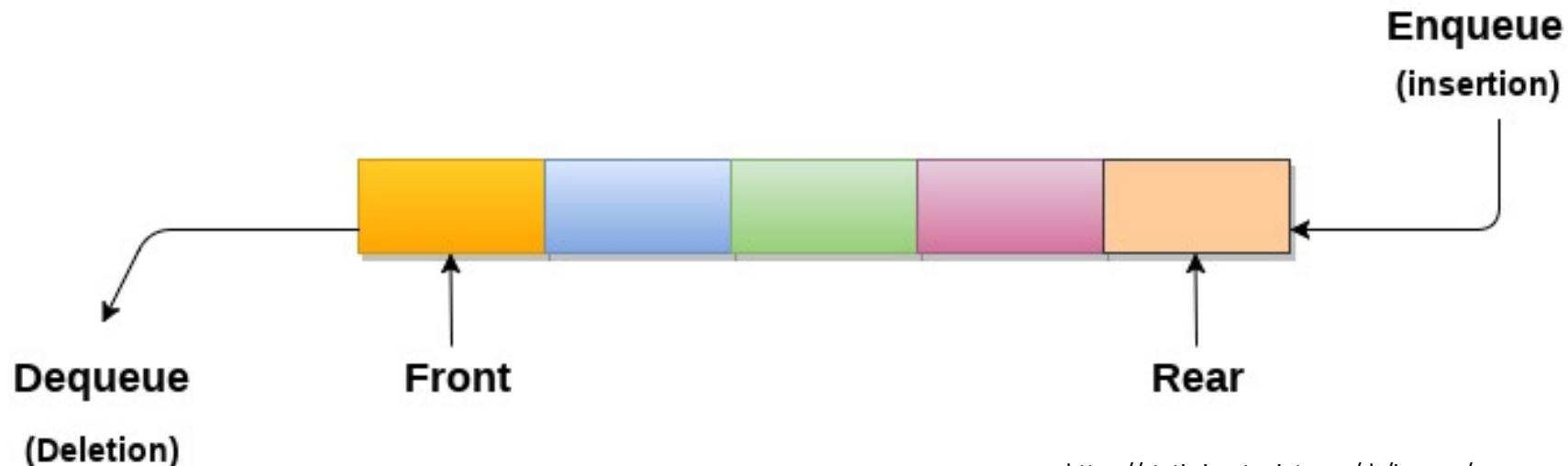
Definisi Queue



- Pada gambar diatas, nilai 1 berada diantrian pertama sebelum nilai 2 maka nilai 1 akan diakses lebih dulu dari sebuah queue, sesuai dengan aturan **FIFO**

Proses pada Queue

- Proses pada queue dilakukan dengan:
 - Menggunakan 2 pointer yaitu **FRONT** dan **REAR**
 - **FRONT** akan menunjuk data elemen pertama dari queue
 - **REAR** akan menunjuk data elemen terakhir dari queue
 - Nilai awal untuk **FRONT** dan **REAR** adalah -1

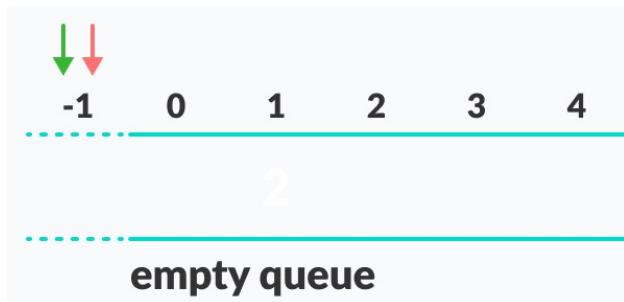
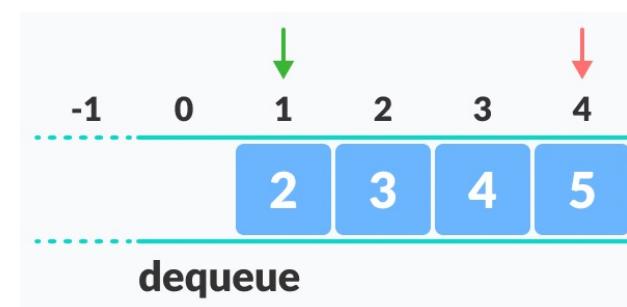
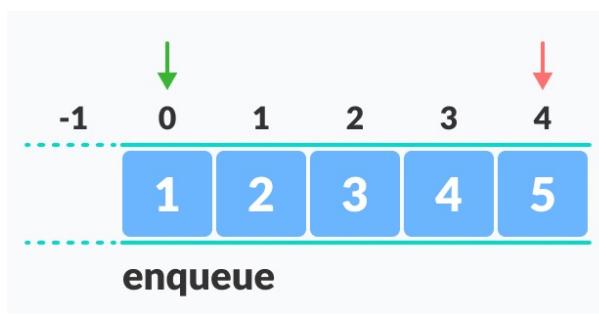
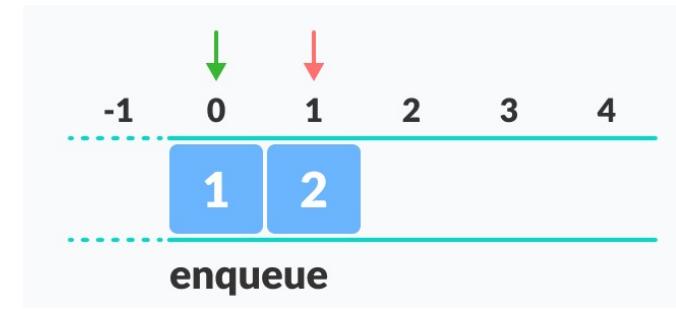
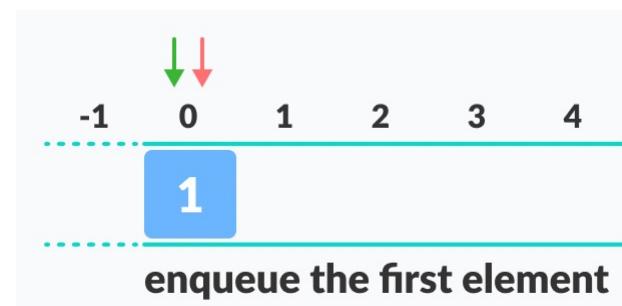


Operasi Pada Queue

Operasi Enqueue() dan Dequeue()

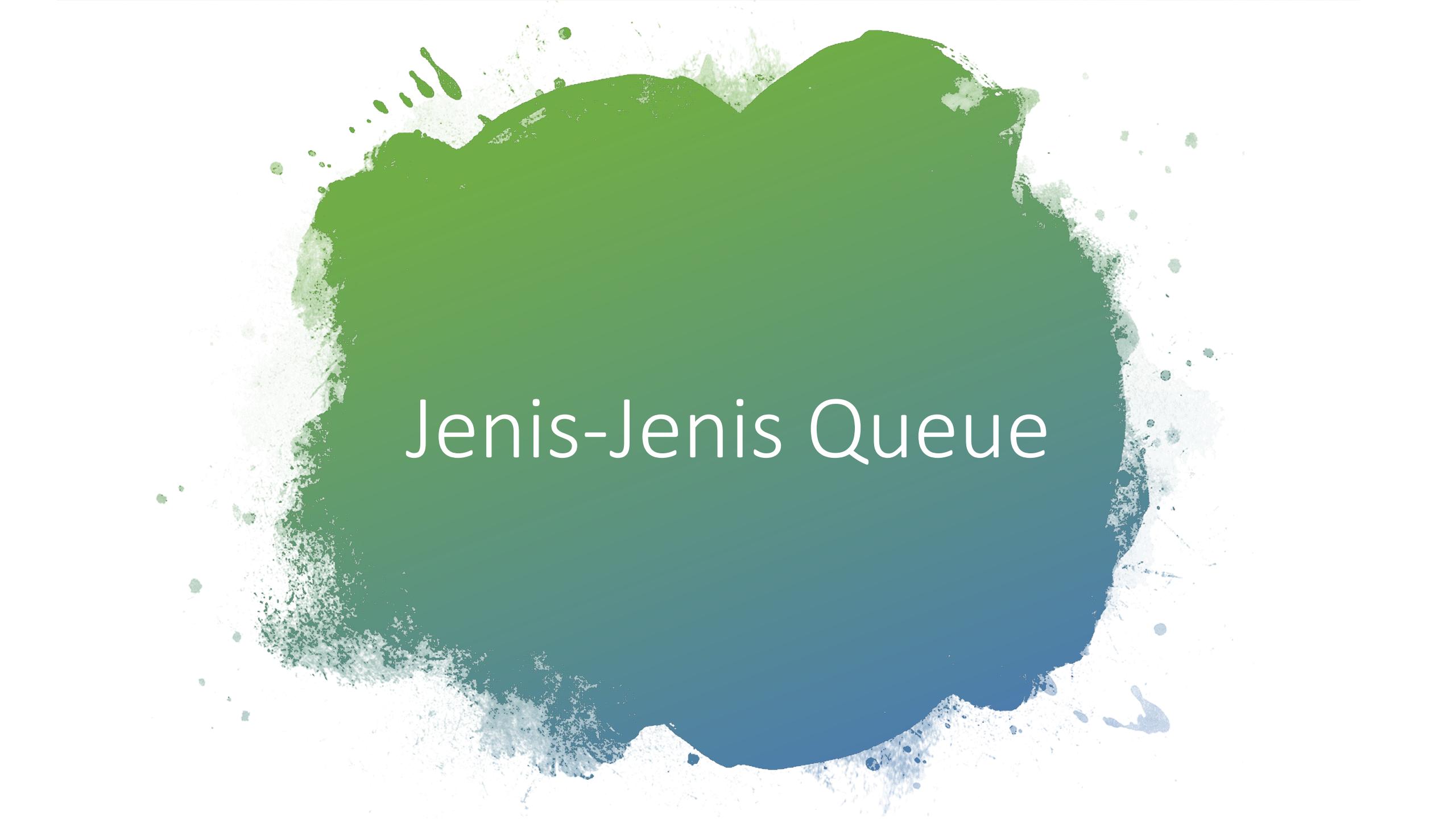
- Operasi **Enqueue** → proses untuk menambahkan data elemen pada bagian akhir (rear) dari sebuah queue. Langkah-langkahnya:
 - Langkah 1: Cek apakah queue sudah penuh
 - Langkah 2: Untuk data elemen pertama, ubah nilai index dari **FRONT** dan **REAR** ke 0
 - Langkah 3: Untuk data elemen selanjutnya, naikkan nilai index **REAR** menjadi 1
 - Langkah 4: Tambahkan data elemen pada posisi yang ditunjuk oleh **REAR**
- Operasi **Dequeue** → proses untuk menghapus data elemen pada bagian depan (front) dari sebuah queue. Langkah-langkahnya:
 - Langkah 1: Cek apakah queue kosong
 - Langkah 2: Kembalikan nilai (elemen) yang ditunjuk oleh **FRONT**
 - Langkah 3: Naikkan nilai index **FRONT** sebanyak 1
 - Langkah 4: Untuk data element terakhir, reset nilai index **FRONT** dan **REAR** menjadi -1

Operasi Enqueue() dan Dequeue



Demo Simple Queue

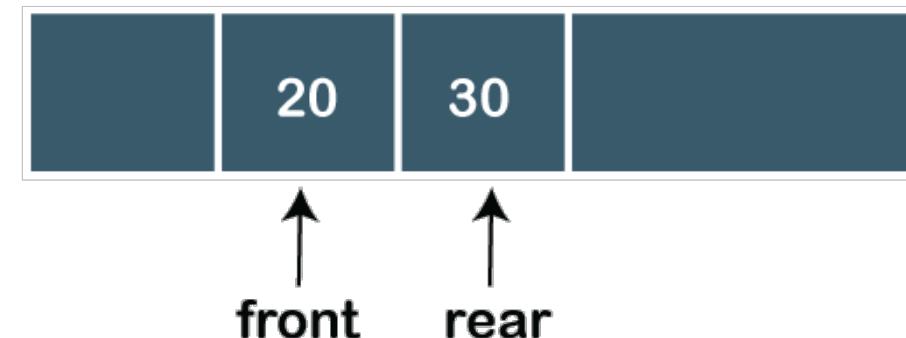
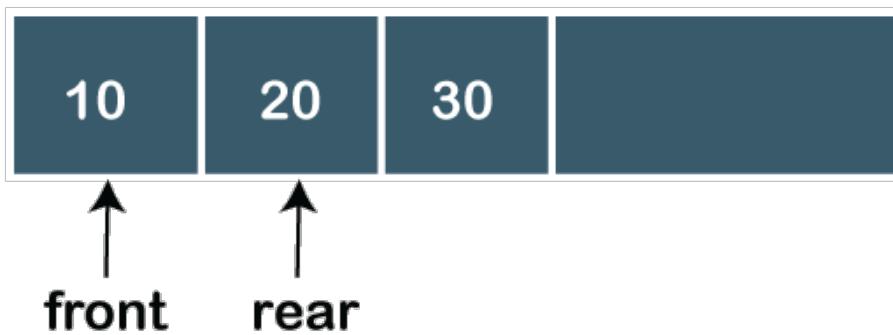
- C
 - <https://www.programiz.com/dsa/queue#c-code>
 - https://www.tutorialspoint.com/data_structures_algorithms/queue_program_in_c.htm
- Java
 - <https://www.programiz.com/dsa/queue#java-code>
- Python
 - <https://www.programiz.com/dsa/queue#python-code>
- C++
 - <https://www.programiz.com/dsa/queue#cpp-code>
- Kompleksitas operasi enqueue() dan dequeue() pada sebuah queue menggunakan array adalah **O(1)**



Jenis-Jenis Queue

Linear Queue (Simple Queue)

- Pada Linear Queue, penambahan data elemen dilakukan dari salah satu bagian queue yaitu **rear** dan penghapusan data elemen dilakukan dari bagian queue lainnya yaitu **front**
- Linear queue sangat mengikuti aturan **FIFO** sebagaimana aturan dari sebuah queue

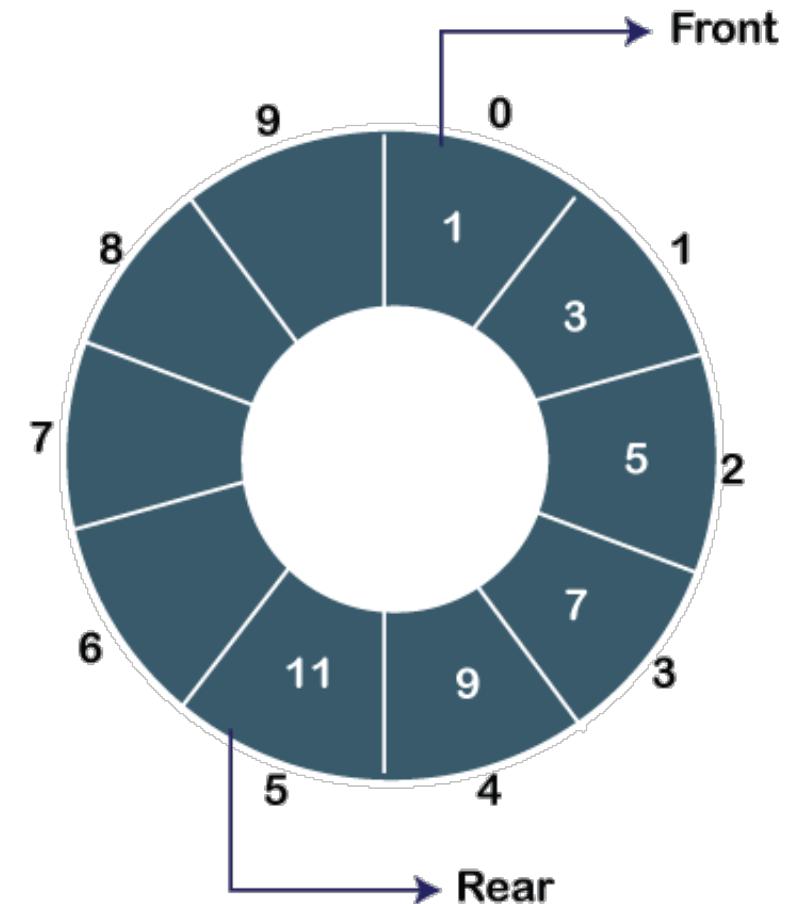


Linear Queue (Simple Queue)

- Pada Linear Queue, jika sebuah data elemen ditambahkan maka nilai indeks dari pointer **REAR** akan dinaikkan sebesar 1
- Sedangkan, jika sebuah data elemen dihapus, maka nilai indeks dari pointer **FRONT** akan dinaikkan sebesar 1
- **Kekurangan dari Linear Queue** adalah proses penambahan data elemen hanya berlaku pada sisi **REAR** saja sehingga jika 3 data elemen dihapus dari queue maka data elemen baru tidak dapat ditambahkan meskipun space masih tersedia di dalam linear queue. Dalam kasus ini, linear queue akan mengalami **overflow** karena pointer **REAR** selalu menunjuk data elemen terakhir dari queue

Circular Queue

- Pada Circular Queue, semua node data elemen direpresentasikan seperti lingkaran, sehingga data elemen terakhir dari queue akan terhubung ke data elemen pertama dari queue tersebut.
- Keuntungan dari Circular Queue adalah **memory utilization** yaitu jika posisi terakhir (**rear**) dari queue penuh dan posisi pertama (**front**) dari queue kosong maka data elemen dapat ditambahkan pada posisi pertama



Circular Queue

Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = FRONT

Write " OVERFLOW "

Goto step 4

[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE IF REAR = MAX - 1 and FRONT != 0

SET REAR = 0

ELSE

SET REAR = (REAR + 1) % MAX

[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Algorithm to delete an element from the circular queue

Step 1: IF FRONT = -1

Write " UNDERFLOW "

Goto Step 4

[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR

SET FRONT = REAR = -1

ELSE

IF FRONT = MAX - 1

SET FRONT = 0

ELSE

SET FRONT = FRONT + 1

[END of IF]

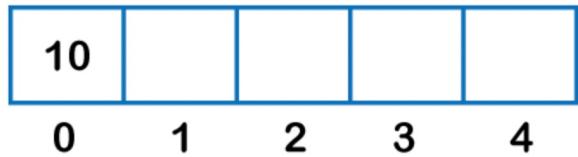
[END OF IF]

Step 4: EXIT

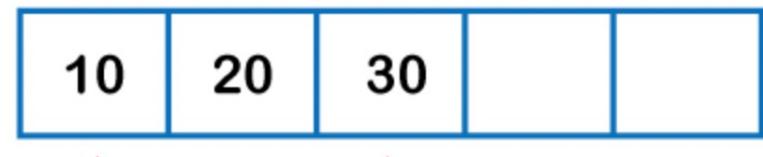
Circular Queue



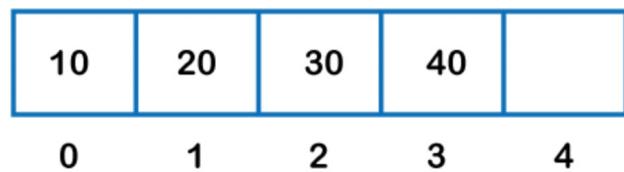
Front = -1
Rear = -1



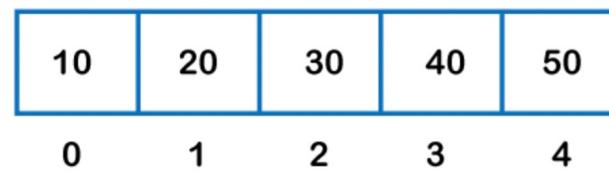
Front = 0
Rear = 0



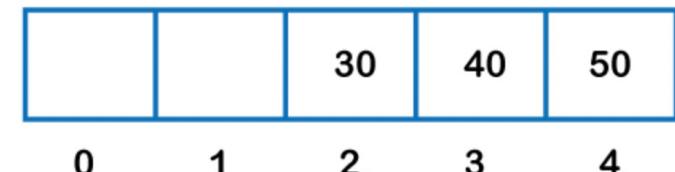
Front = 0
Rear = 2



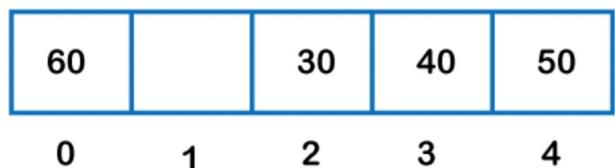
Front = 0
Rear = 3



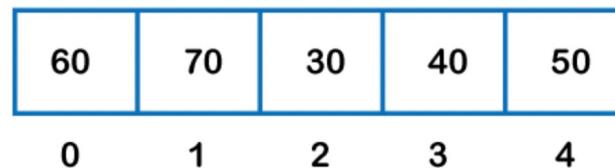
Front = 0
Rear = 4



Front = 2
Rear = 4
dequeue



Rear
Front



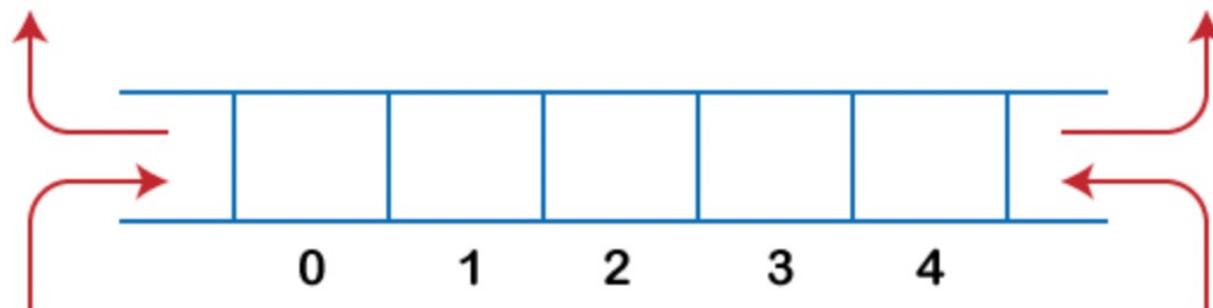
Rear
Front

Demo Circular Queue

<https://www.javatpoint.com/circular-queue>

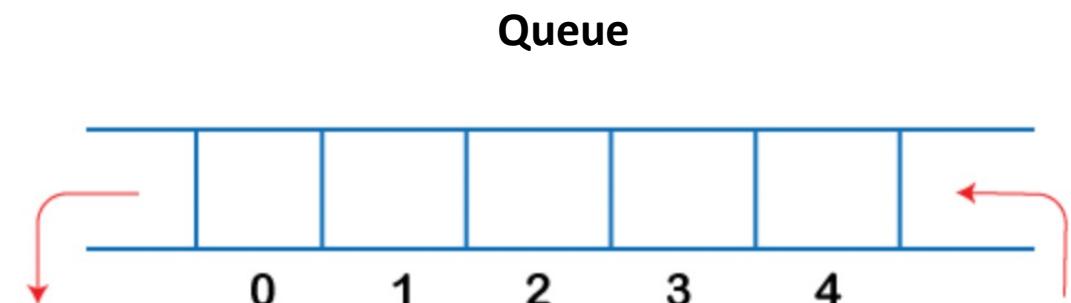
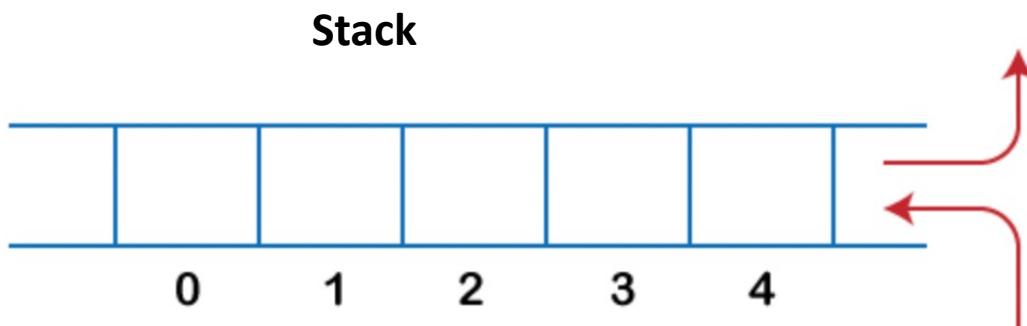
Deque (Double Ended) Queue

- Pada simple queue, penambahan data elemen dilakukan pada bagian **rear**, sedangkan penghapusan data elemen dilakukan pada bagian **front**
- Pada Deque (Double Ended) Queue, penambahan dan penghapusan data elemen dilakukan pada kedua sisi queue. Dengan kata lain, deque merupakan generalisasi dari sebuah queue
- Deque dapat digunakan sebagai **stack** atau **queue** karena penambahan dan penghapusan data dapat dilakukan pada kedua sisi



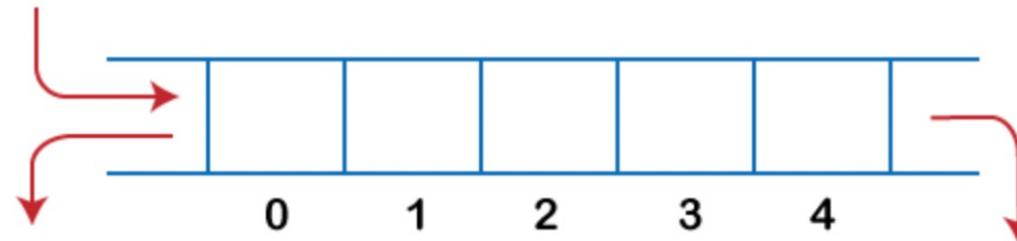
Deque (Double Ended) Queue

- Pada deque, penambahan dan penghapusan data elemen dapat dilakukan pada satu sisi saja seperti pada **stack**. Sama halnya, penambahan data elemen juga dapat dilakukan pada satu sisi dan penghapusan data elemen pada sisi lainnya seperti pada **queue**

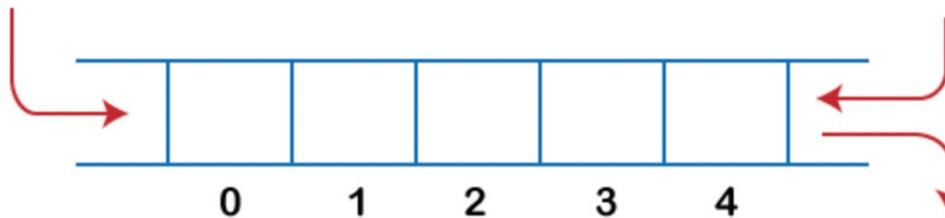


Deque (Double Ended) Queue

- **Input-restricted queue** → penambahan data elemen hanya dilakukan pada salah satu sisi saja, sedangkan penghapusan data elemen dapat dilakukan pada kedua sisi



- **Output-restricted queue** → penghapusan data elemen hanya dilakukan pada salah satu sisi saja, sedangkan penambahan data elemen dapat dilakukan pada kedua sisi

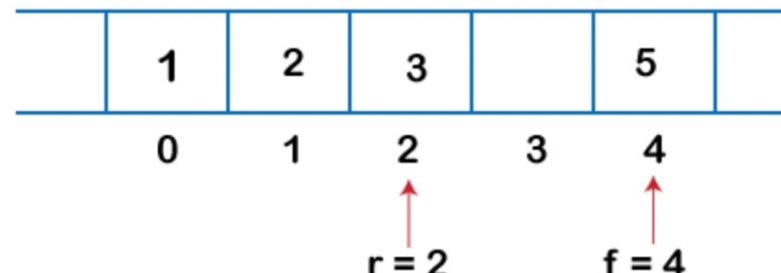
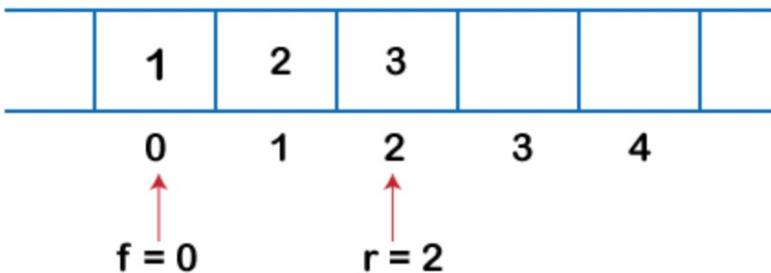
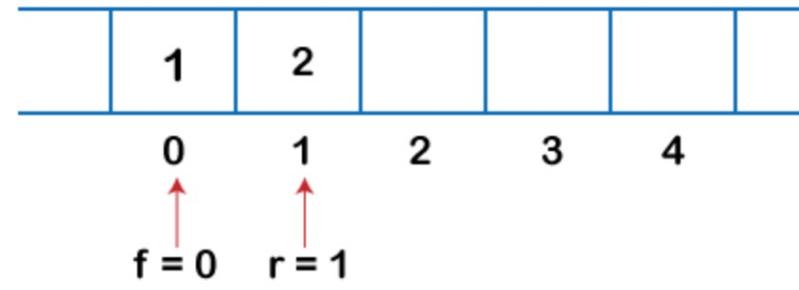
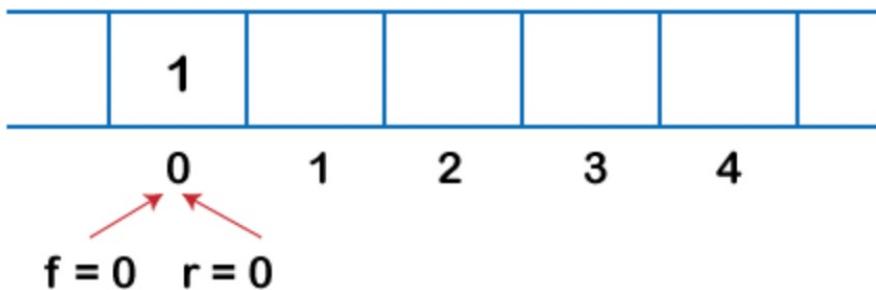


Deque (Double Ended) Queue

- Deque dapat diimplementasikan menggunakan **circular array** dan **doubly linkedlist**
- **Circular array** adalah sebuah array yang bagian akhir data elemen akan terhubung ke bagian pertama data elemen sehingga tidak akan terjadi **overflow**
- **Operasi pada deque**
 - Insert at front
 - Delete from front
 - Insert at rear
 - Delete from rear

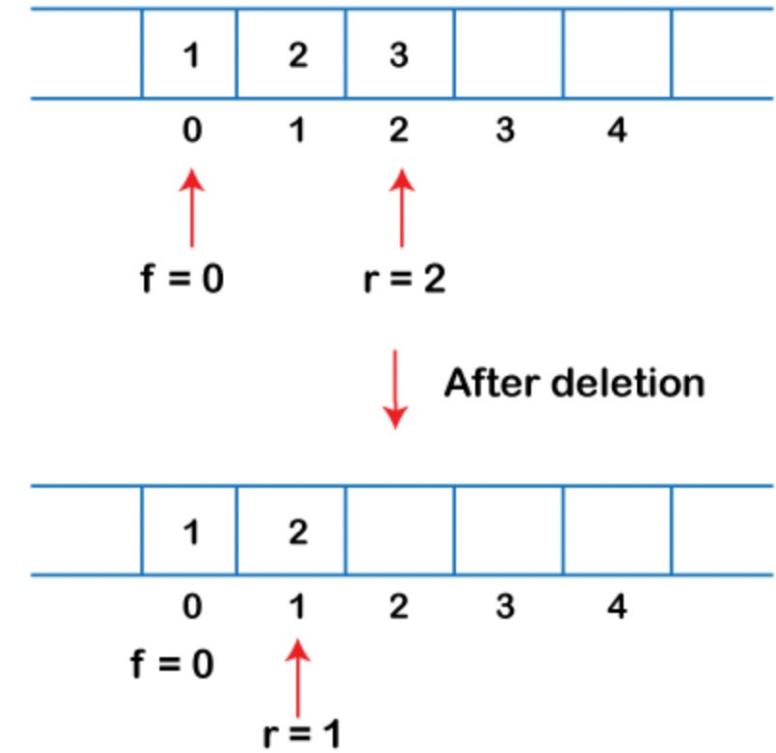
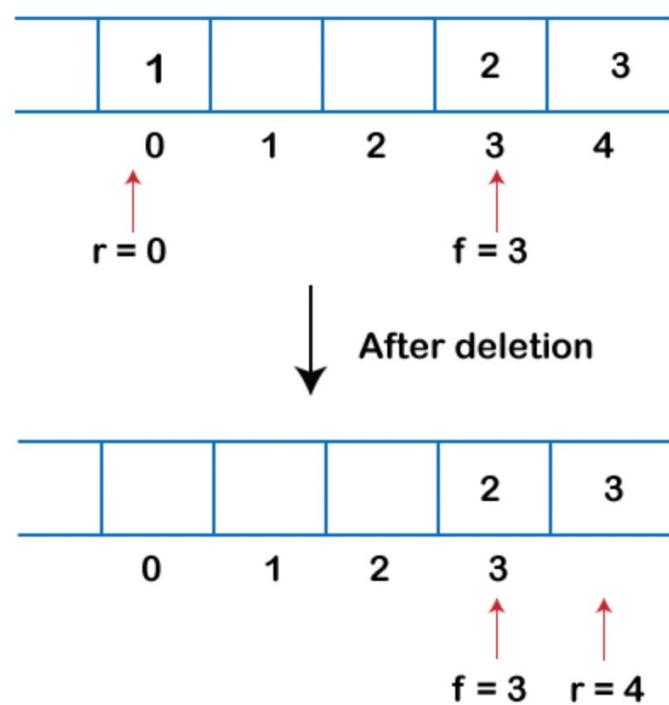
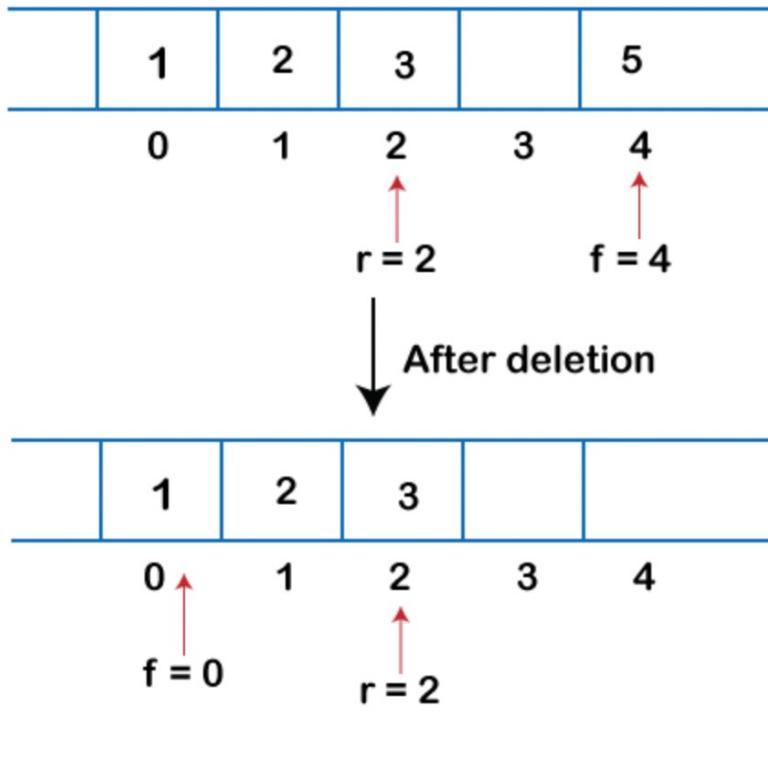
Deque (Double Ended) Queue

- Operasi Enqueue



Deque (Double Ended) Queue

- Operasi Dequeue



Demo Deque Queue

<https://www.javatpoint.com/ds-deque>

Q & A

Referensi

- [https://www.tutorialspoint.com/data structures algorithms/dsa queue.htm](https://www.tutorialspoint.com/data_structures_algorithms/dsa_queue.htm)
- <https://www.javatpoint.com/data-structure-queue>
- <https://www.programiz.com/dsa/queue>

INF218

Struktur Data & Algoritma

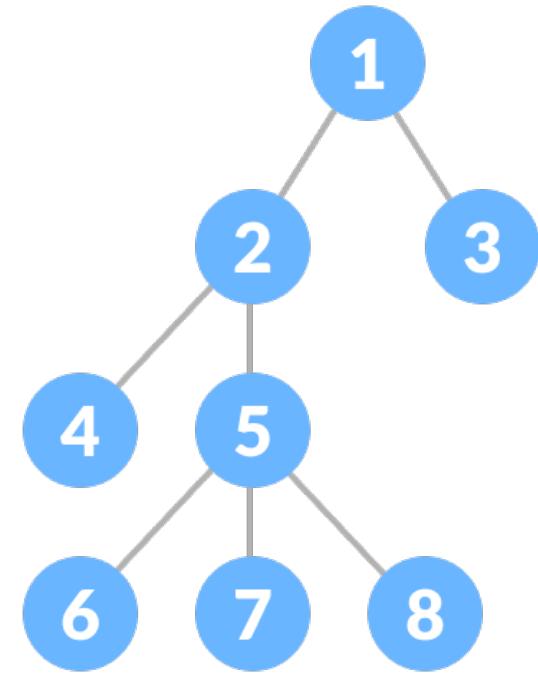
Struktur Data Tree

Alim Misbullah, S.Si., M.S.

Definisi Tree

Definisi Tree

- **Tree** adalah non-linear hirarki struktur data yang terdiri dari **node** yang saling terhubung melalui **edge**
- Mengapa perlu tree?
 - Struktur data yang lain seperti array, linkedlist, stack dan queue merupakan linear struktur data yang datanya saling terhubung secara berurutan
 - Pada linear struktur data, penambahan data yang banyak akan membutuhkan waktu yang banyak pula sehingga tidak bisa digunakan pada komputasi saat ini
 - Tree menjadi solusi karena memungkinkan untuk mengakses data secara cepat dan mudah karena tree merupakan non-linear struktur data



Tree

Terminologi Tree

- **Nodes**

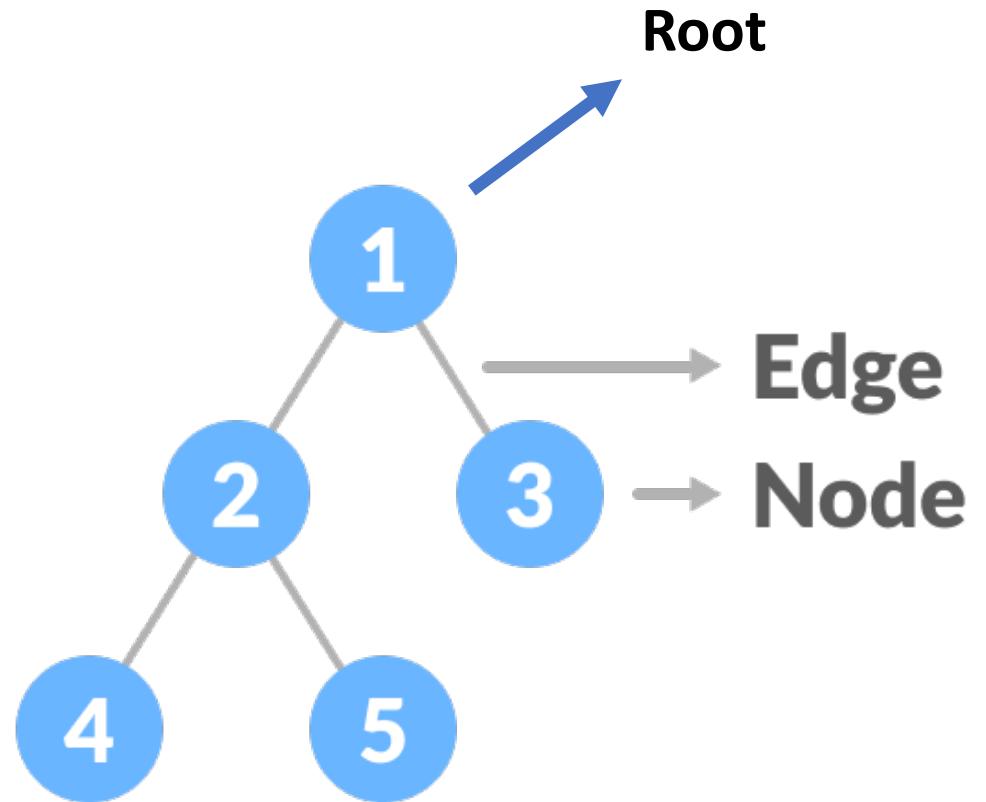
- Sebuah entitas yang mengandung key or value dan pointer ke nodes dibawahnya (child nodes)
- Node yang terakhir dari setiap path disebut **leaf nodes or external nodes** yang tidak memiliki hubungan (link)/pointer ke child node
- Node yang memiliki setidaknya satu child nodes disebut **internal node**

- **Edge**

- Link/penghubung antara dua node

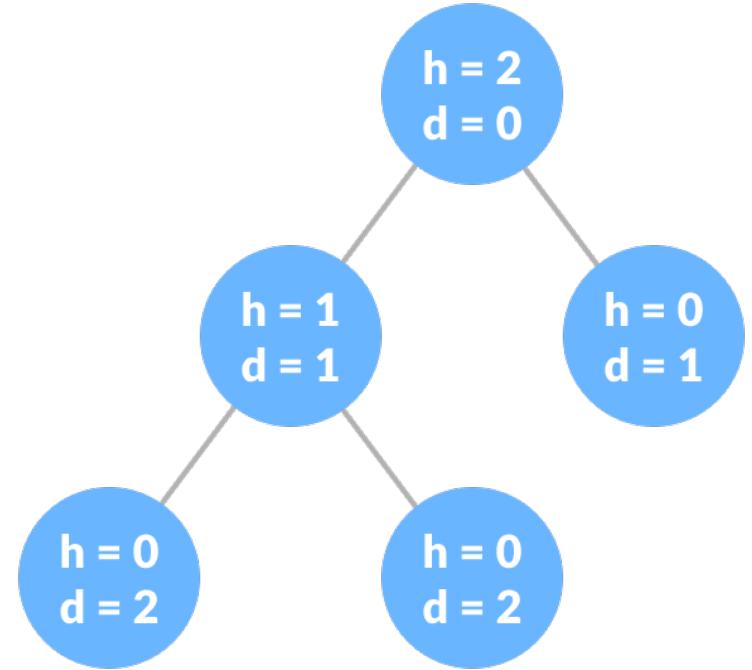
- **Root**

- Node yang teratas dari sebuah tree



Terminologi Tree

- **Height of a node**
 - Jumlah edges dari sebuah node ke node terakhir di bawahnya (leaf) atau path terjauh dari node ke leaf
- **Depth of a node**
 - Jumlah edges dari root ke node tersebut
- **Height of a tree**
 - Tinggi dari sebuah root node atau kedalaman dari sebuah node
- **Degree of a node**
 - Jumlah total percabangan dari sebuah node



Height and depth of each node in a tree

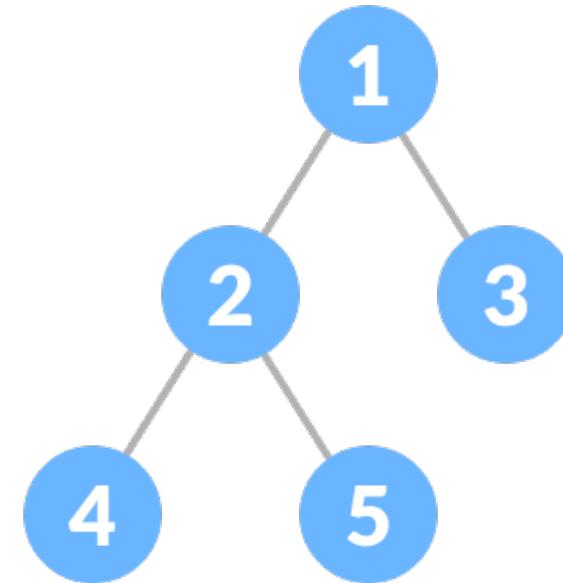
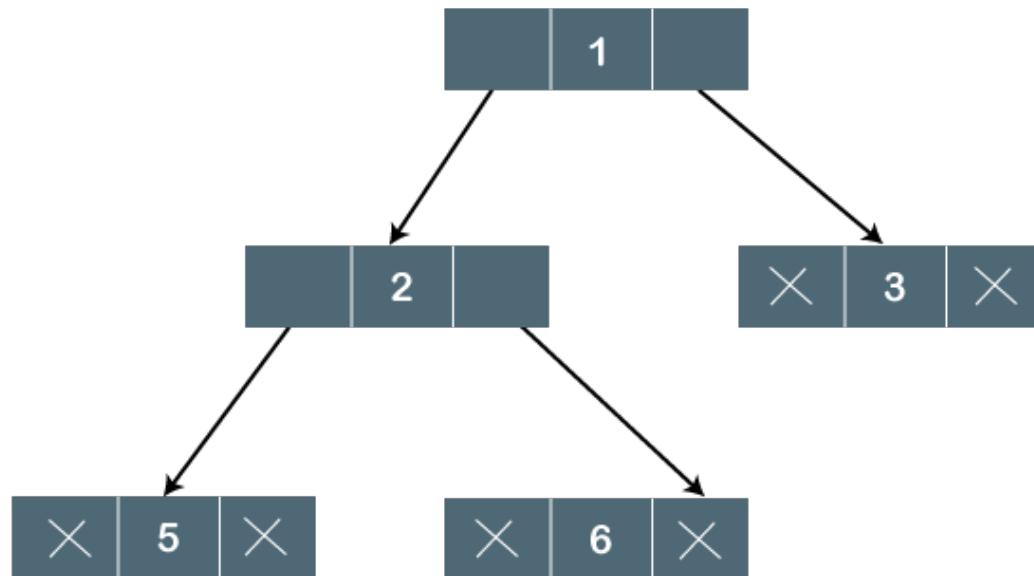
Jenis-jenis Tree

- **Binary Tree**
 - Sebuah struktur data tree yang setiap parent node dapat mempunyai paling banyak 2 children nodes
- **Binary Search Tree**
 - Sebuah struktur data tree yang memiliki aturan seperti binary tree, namun BST disebut sebagai ordered binary tree sehingga sangat efisien untuk proses pencarian
- **AVL Tree**
 - Sebuah self-balancing binary search tree yang mana setiap node akan menjaga sebuah balance factor diantara nilai -1, 0, dan 1
- **B-Tree**
 - Sebuah self-balancing BST yang mana setiap node dapat mengandung lebih dari 1 key dan lebih dari 2 children nodes. B-Tree ada bentuk umum dari BST

Binary Tree

Binary Tree

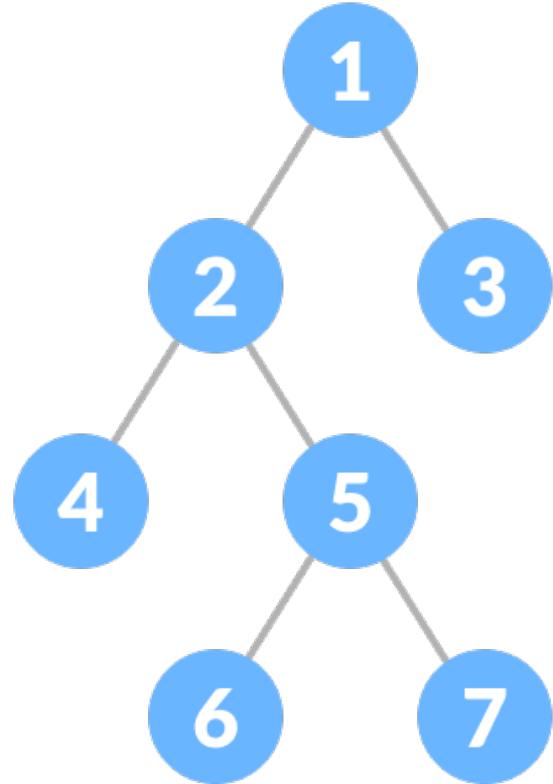
- Sebuah tree yang setiap parent node memiliki paling banyak 2 (dua) child node
- Dinamai BT karena setiap node dapat memiliki 0, 1 atau 2 children



Binary Tree

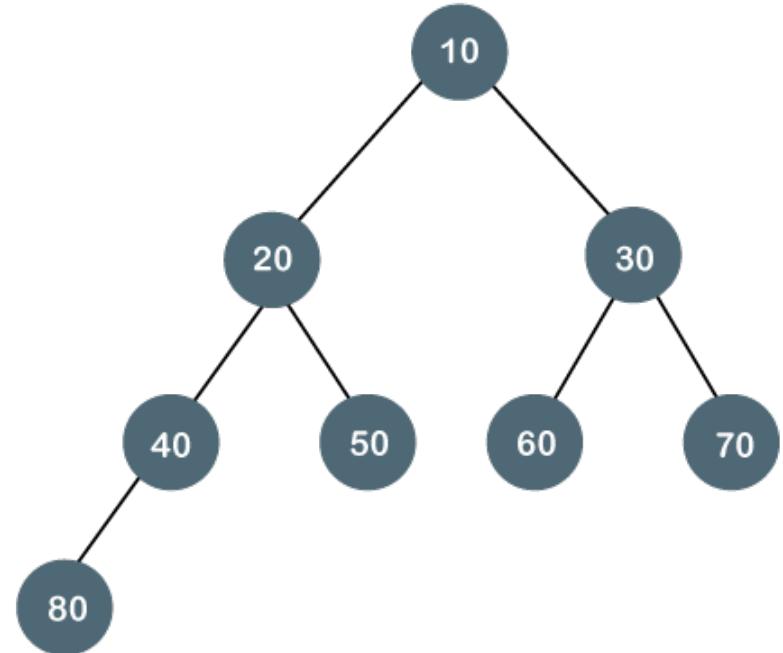
Jenis Binary Tree

- Full Binary Tree
 - Jenis binary tree yang setiap parent/internal node mempunyai 2 (dua) atau tidak memiliki children sama sekali
 - Tree ini juga disebut dengan **proper binary tree**
- Theorema Full Binary Tree
 - <https://www.programiz.com/dsa/full-binary-tree>
- Implementation in C
 - <https://www.programiz.com/dsa/full-binary-tree#c-code>



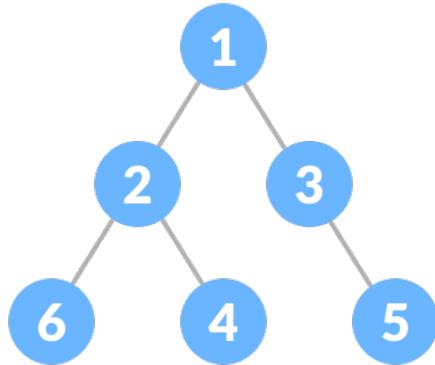
Jenis Binary Tree

- Complete Binary Tree
 - Jenis binary tree yang semua node harus terisi di setiap level, terkecuali level terakhir
 - Pada level terakhir, semua nodes harus sebisa mungkin bersandar ke kiri
 - Pada complete binary tree, node harus ditambahkan mulai dari kiri
- Implementation
 - <https://www.programiz.com/dsa/complete-binary-tree#c-code>

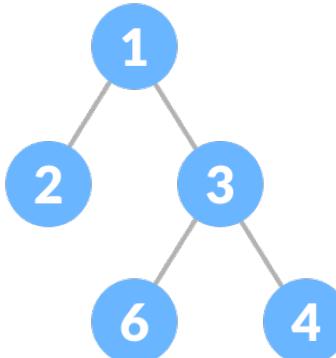


Jenis Binary Tree

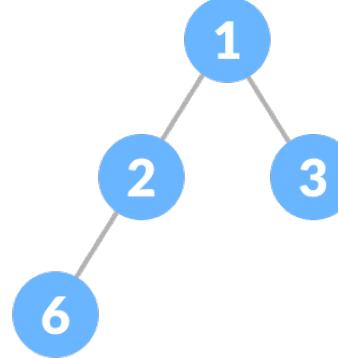
- Perbandingan Complete Binary Tree dan Full Binary Tree



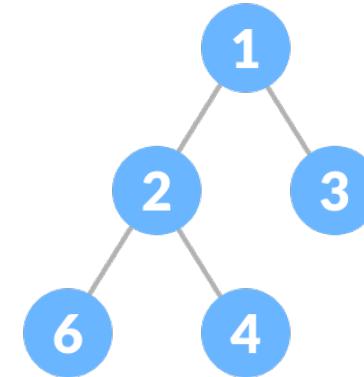
Full Binary Tree
 Complete Binary Tree



Full Binary Tree
 Complete Binary Tree



Full Binary Tree
 Complete Binary Tree



Full Binary Tree
 Complete Binary Tree

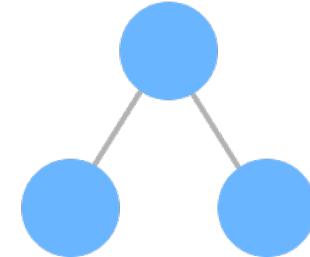
Jenis Binary Tree

- Perfect Binary Tree
 - Jenis binary tree yang setiap internal node harus mempunyai 2 (dua) children dan leaf node pada level yang sama
 - Jika tree tidak memiliki children, maka disebut perfect binary tree dengan $h = 0$

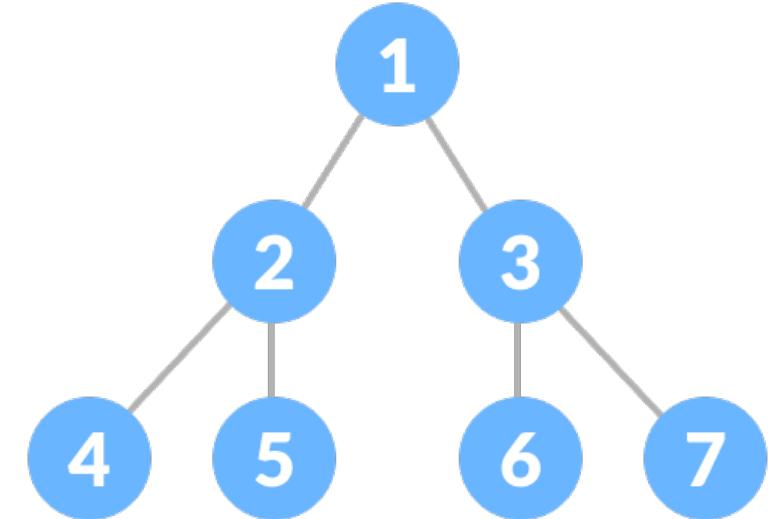
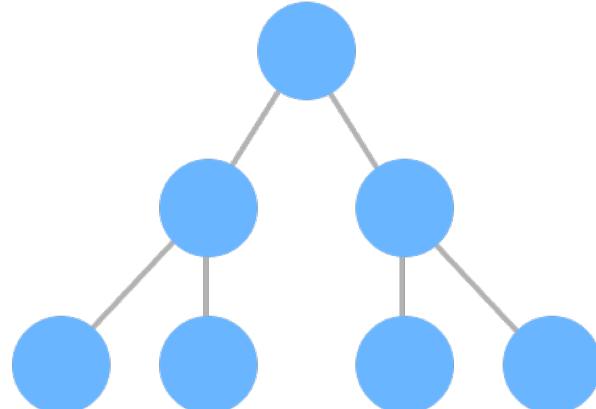
tree-1



tree-2



tree-3

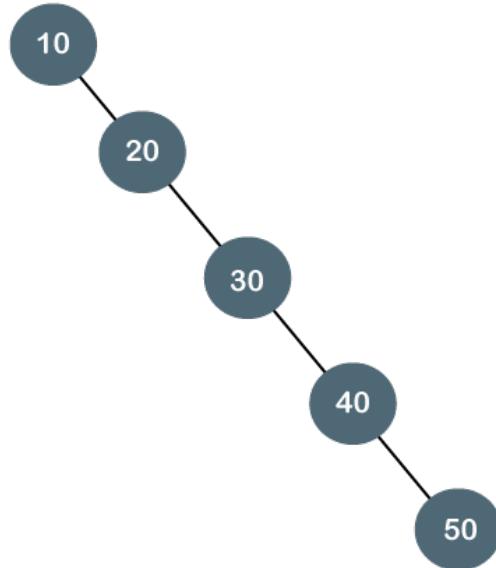


Implementation:

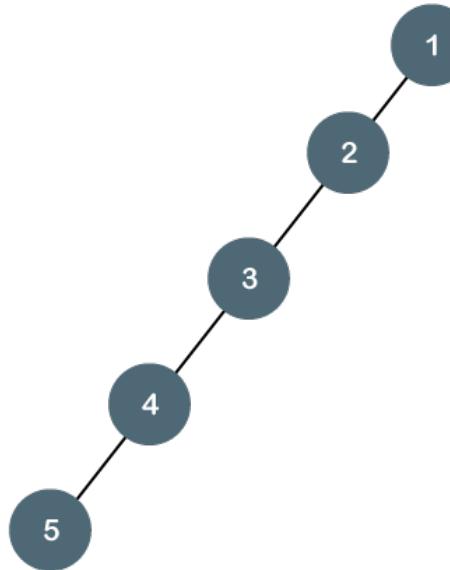
<https://www.programiz.com/dsa/perfect-binary-tree#c-code>

Jenis Binary Tree

- Degenerate/Pathological Binary Tree
 - Jenis binary tree yang hanya memiliki satu children baik pada sebelah kiri atau kanan



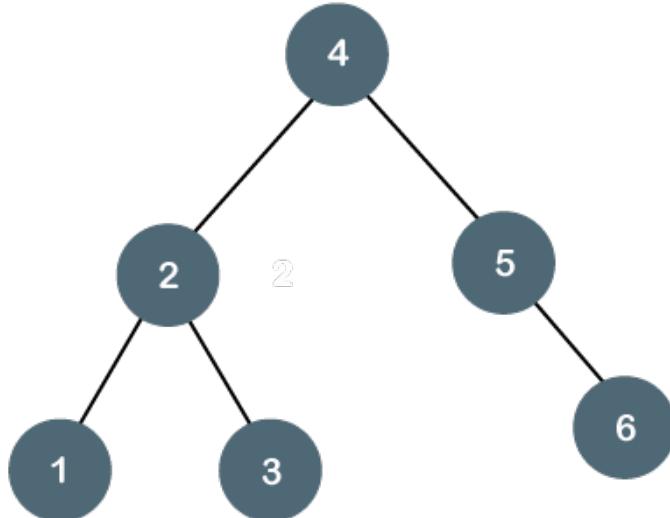
Right-skewed or right child only



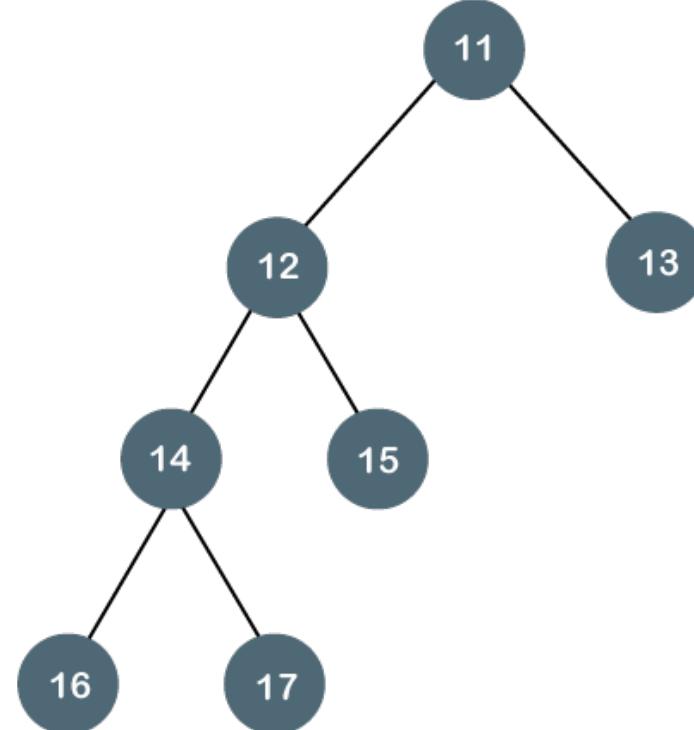
Left-skewed or left child only

Jenis Binary Tree

- Balance Binary Tree
 - Jenis binary tree yang height bagian kiri dan kanan dari tree berbeda maksimum 1



Balance Binary Tree
 $H(L) - H(R) = 0$

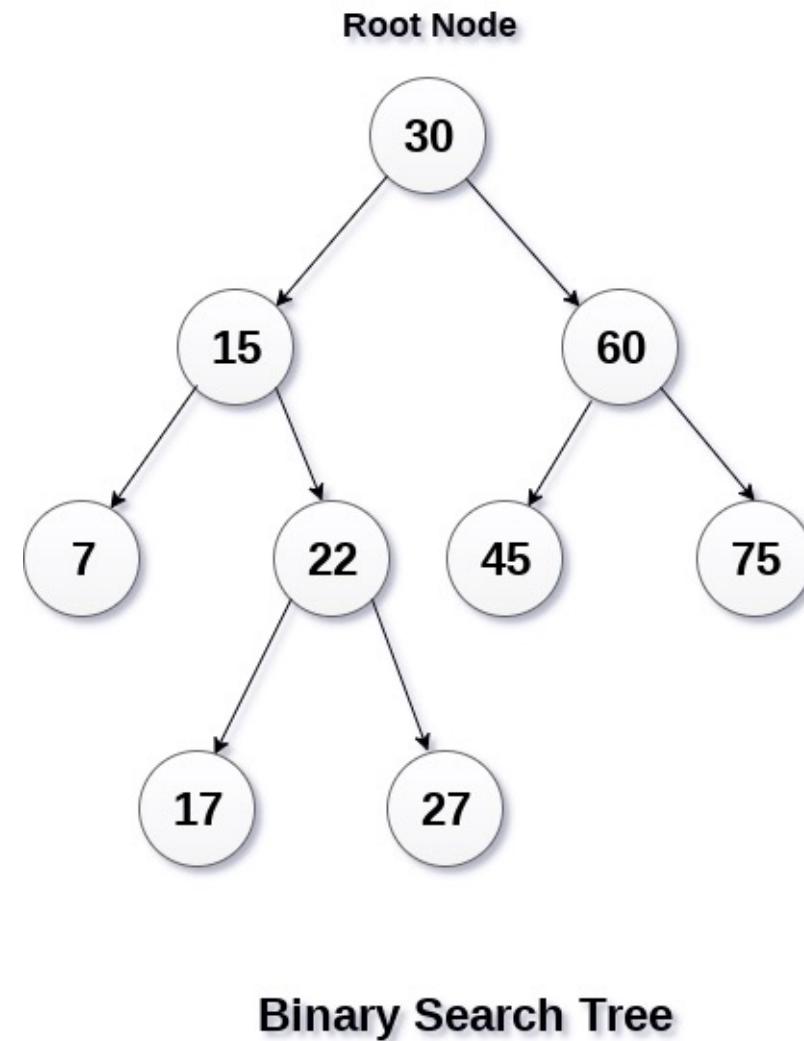


Bukan Balance Binary Tree
 $H(L) - H(R) = 2$

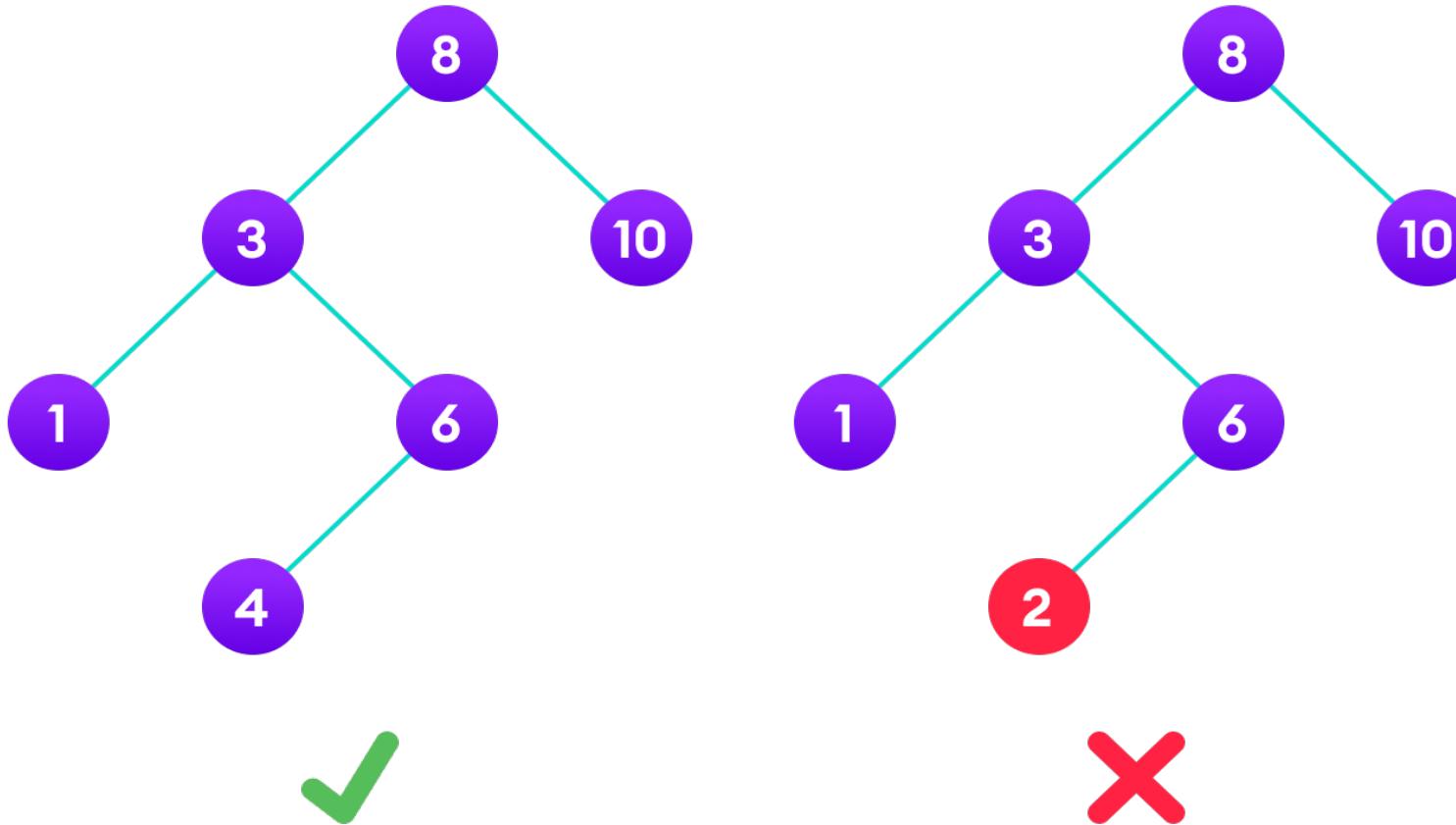
Binary Search Tree

Binary Search Tree

- Binary search tree merupakan sebuah struktur data yang mengizinkan kita secara cepat untuk melakukan pengurutan dari susunan bilangan atau disebut juga dengan **ordered binary tree**
- Pada binary search tree, **nilai dari semua node pada bagian kiri < nilai root**
- Sementara, **nilai dari semua node pada bagian kanan > nilai root**

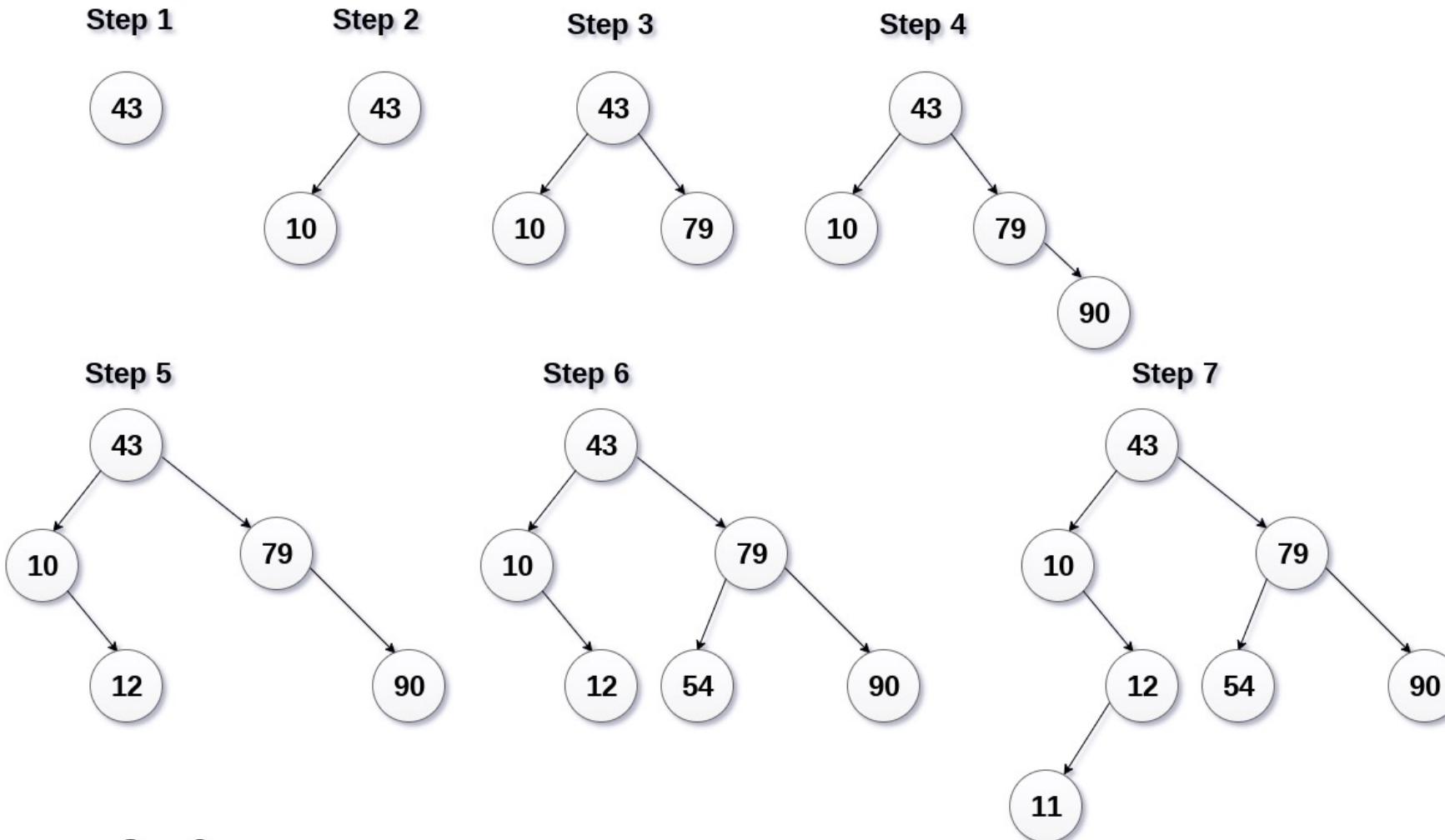


Binary Search Tree

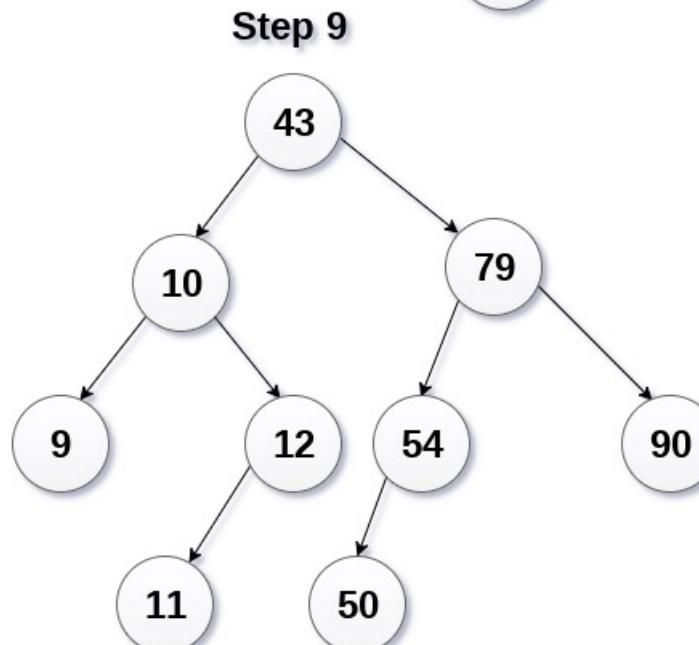
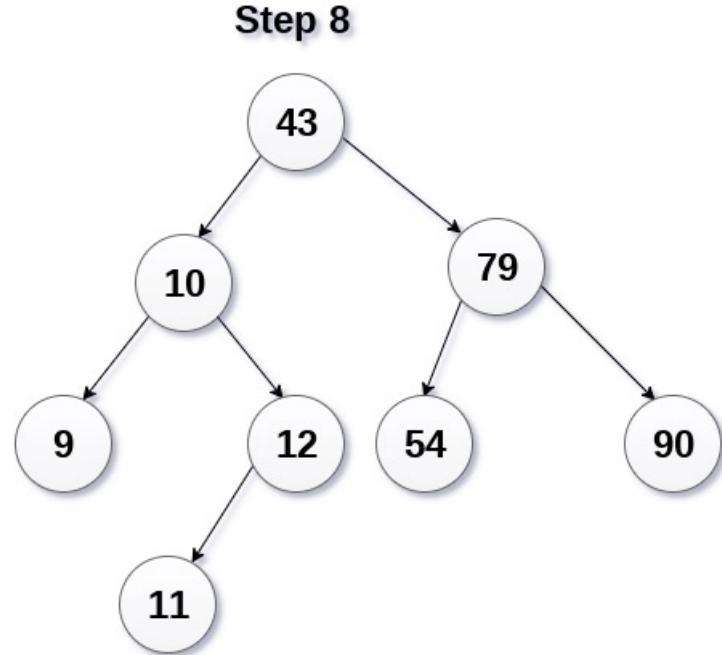


Bagian tree sebelah kanan adalah bukan binary search tree karena terdapat nilai lebih kecil di sebelah kanan “3” yaitu “2”

Proses membuat BST



Proses membuat BST



Operasi pada Binary Search Tree

- Search Operation (Operasi Pencarian)
 - Algoritmanya

```
If root == NULL  
    return NULL;  
If number == root->data  
    return root->data;  
If number < root->data  
    return search(root->left)  
If number > root->data  
    return search(root->right)
```

Ilustrasi:

<https://www.javatpoint.com/searching-in-binary-search-tree>

- Insert Operation (Operasi Penambahan)
 - Algoritmanya

```
If node == NULL  
    return createNode(data)  
if (data < node->data)  
    node->left = insert(node->left, data);  
else if (data > node->data)  
    node->right = insert(node->right, data);  
return node;
```

Ilustrasi:

<https://www.javatpoint.com/insertion-in-binary-search-tree>

Operasi pada Binary Search Tree

- Deletion Operation (Operasi Penghapusan)
 - Algoritmanya

Delete (TREE, ITEM)

- Step 1: IF TREE = NULL

 Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA

 Delete(TREE->LEFT, ITEM)

 ELSE IF ITEM > TREE -> DATA

 Delete(TREE -> RIGHT, ITEM)

 ELSE IF TREE -> LEFT AND TREE -> RIGHT

 SET TEMP = findLargestNode(TREE -> LEFT)

 SET TREE -> DATA = TEMP -> DATA

 Delete(TREE -> LEFT, TEMP -> DATA)

 ELSE

 SET TEMP = TREE

 IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL

 SET TREE = NULL

 ELSE IF TREE -> LEFT != NULL

 SET TREE = TREE -> LEFT

 ELSE

 SET TREE = TREE -> RIGHT

 [END OF IF]

 FREE TEMP

 [END OF IF]

- Step 2: END

Ilustrasi:

<https://www.javatpoint.com/deletion-in-binary-search-tree>

Q & A

Referensi

- <https://www.programiz.com/dsa/trees>
- <https://www.programiz.com/dsa/binary-search-tree>
- <https://www.javatpoint.com/binary-search-tree>

INF218

Struktur Data & Algoritma

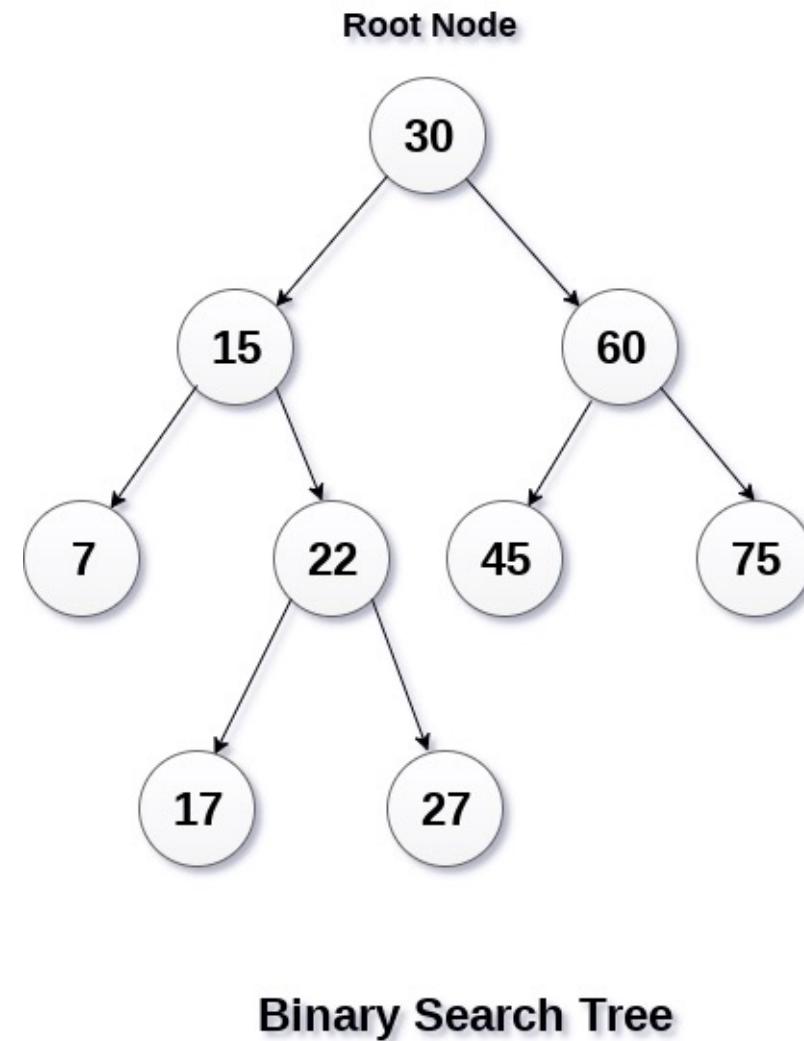
BST dan Tree Traversal

Alim Misbullah, S.Si., M.S.

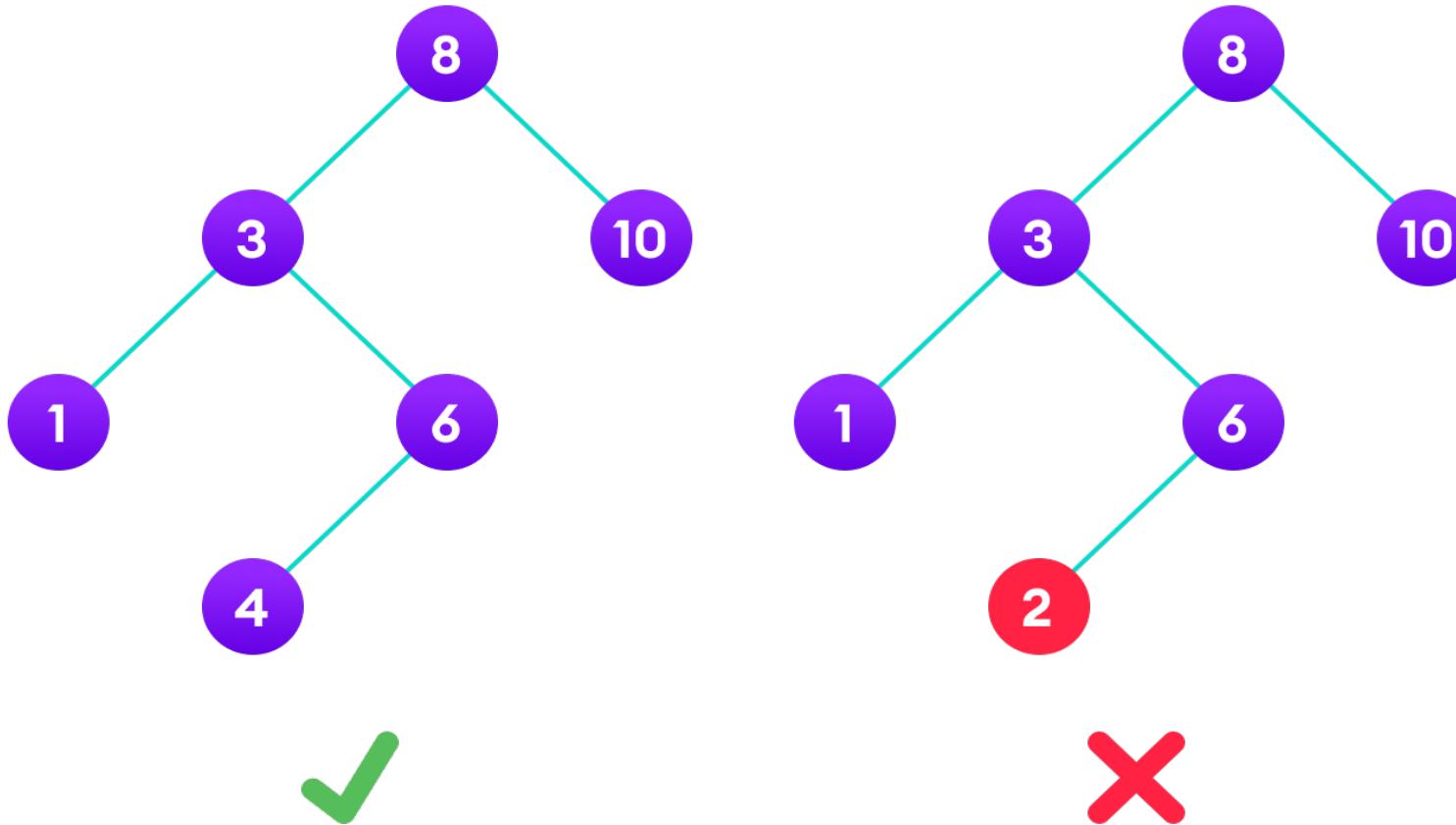
Binary Search Tree

Binary Search Tree

- Binary search tree merupakan sebuah struktur data yang mengizinkan kita secara cepat untuk melakukan pengurutan dari susunan bilangan atau disebut juga dengan **ordered binary tree**
- Pada binary search tree, **nilai dari semua node pada bagian kiri < nilai root**
- Sementara, **nilai dari semua node pada bagian kanan > nilai root**



Binary Search Tree



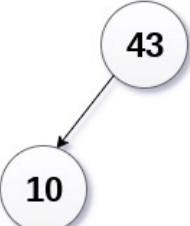
Bagian tree sebelah kanan adalah bukan binary search tree karena terdapat nilai lebih kecil di sebelah kanan “3” yaitu “2”

Proses membuat BST

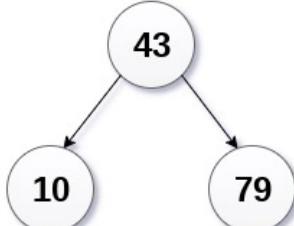
Step 1



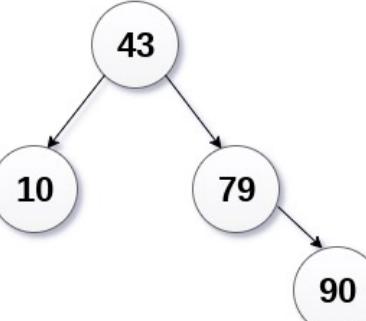
Step 2



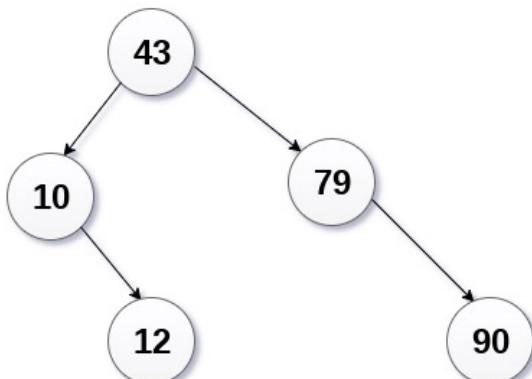
Step 3



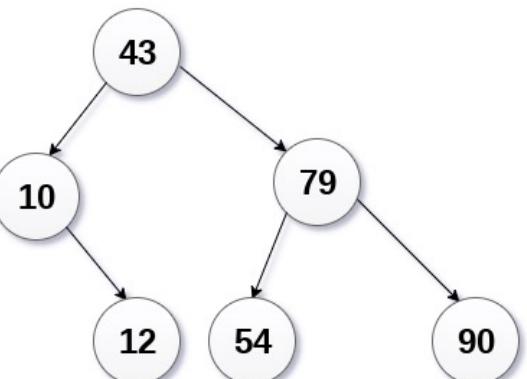
Step 4



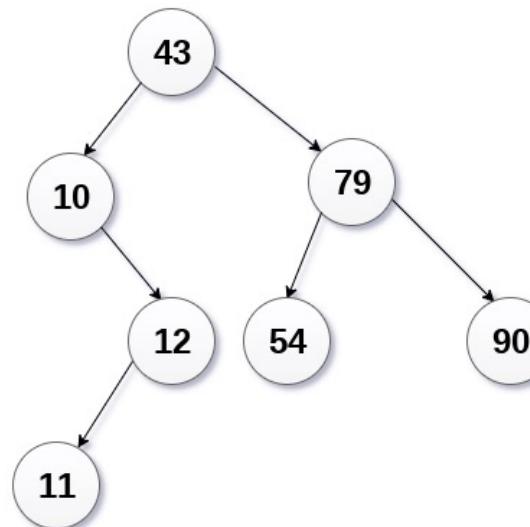
Step 5



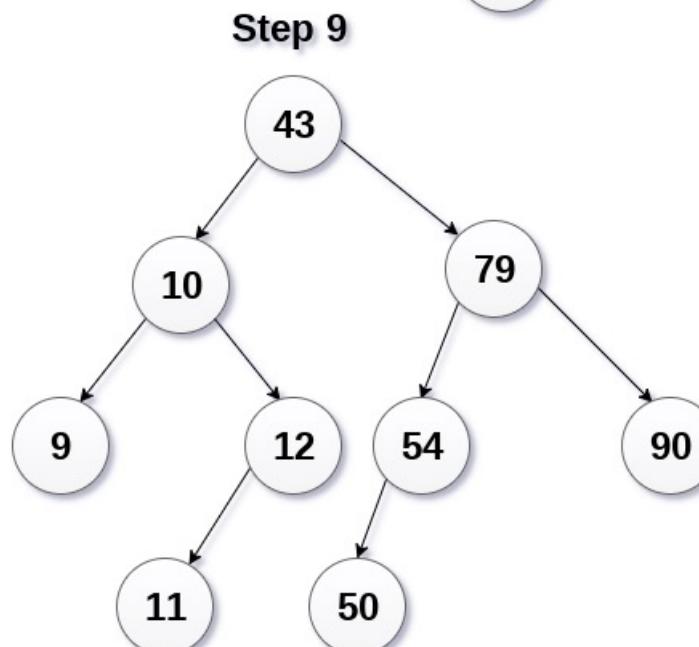
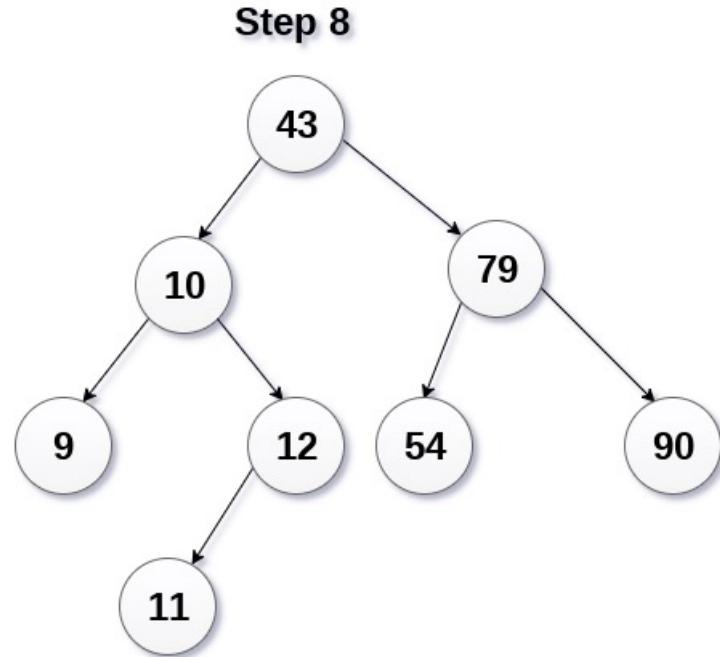
Step 6



Step 7



Proses membuat BST



Operasi pada Binary Search Tree

- Search Operation (Operasi Pencarian)
 - Algoritmanya

```
If root == NULL  
    return NULL;  
If number == root->data  
    return root->data;  
If number < root->data  
    return search(root->left)  
If number > root->data  
    return search(root->right)
```

Ilustrasi:

<https://www.javatpoint.com/searching-in-binary-search-tree>

- Insert Operation (Operasi Penambahan)
 - Algoritmanya

```
If node == NULL  
    return createNode(data)  
if (data < node->data)  
    node->left = insert(node->left, data);  
else if (data > node->data)  
    node->right = insert(node->right, data);  
return node;
```

Ilustrasi:

<https://www.javatpoint.com/insertion-in-binary-search-tree>

Operasi pada Binary Search Tree

- Deletion Operation (Operasi Penghapusan)
 - Algoritmanya

Delete (TREE, ITEM)

- Step 1: IF TREE = NULL

 Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA

 Delete(TREE->LEFT, ITEM)

 ELSE IF ITEM > TREE -> DATA

 Delete(TREE -> RIGHT, ITEM)

 ELSE IF TREE -> LEFT AND TREE -> RIGHT

 SET TEMP = findLargestNode(TREE -> LEFT)

 SET TREE -> DATA = TEMP -> DATA

 Delete(TREE -> LEFT, TEMP -> DATA)

 ELSE

 SET TEMP = TREE

 IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL

 SET TREE = NULL

 ELSE IF TREE -> LEFT != NULL

 SET TREE = TREE -> LEFT

 ELSE

 SET TREE = TREE -> RIGHT

 [END OF IF]

 FREE TEMP

 [END OF IF]

- Step 2: END

Ilustrasi:

<https://www.javatpoint.com/deletion-in-binary-search-tree>

Tree Traversal

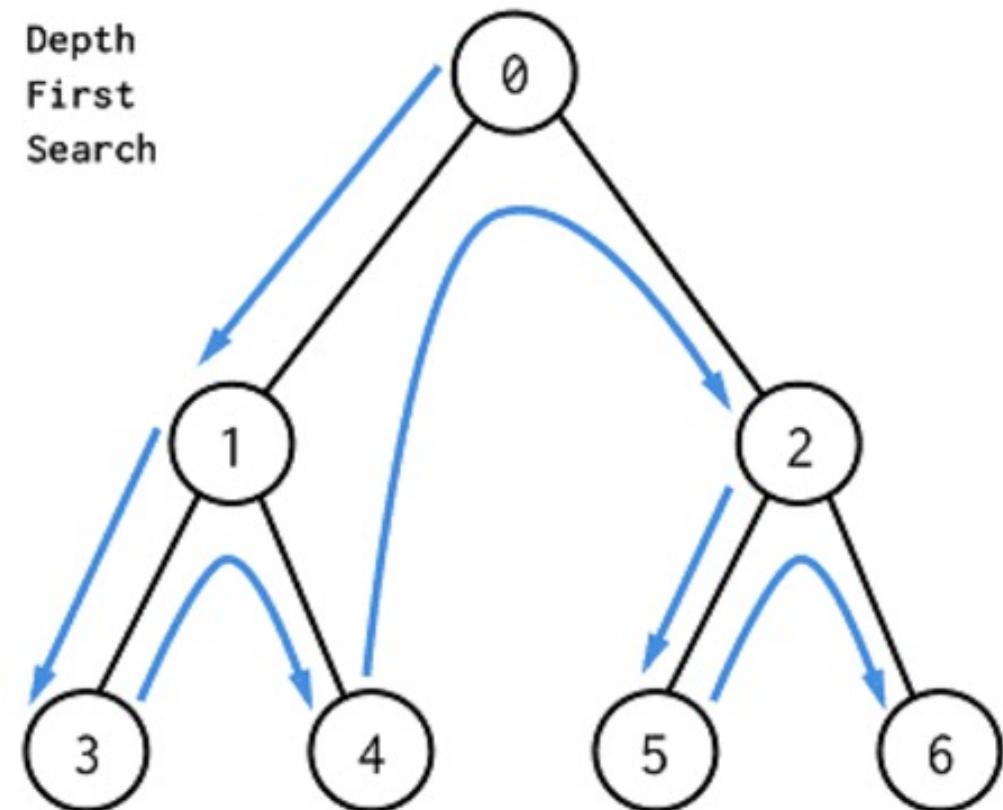
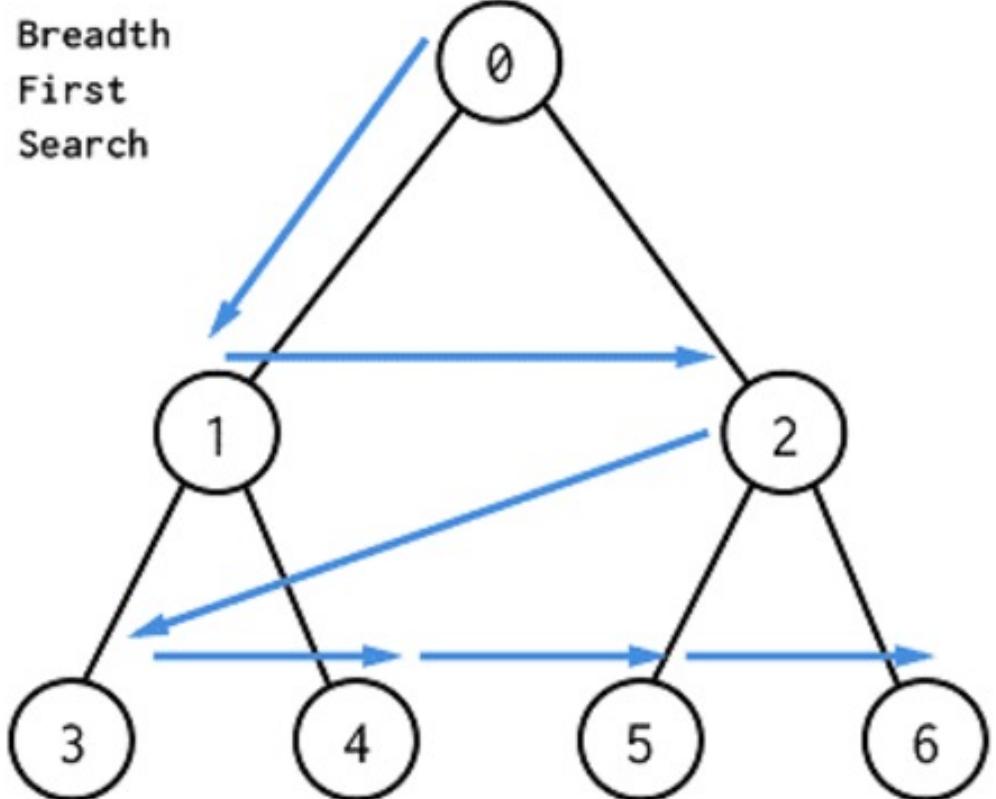
Definisi Tree Traversal

- Traversal maksudnya adalah **penelurusan**
- Tree Traversal adalah menelusuri setiap node yang ada pada sebuah tree
- Misalnya, kita ingin menambahkan data atau mencari data yang ada di dalam tree, maka kita perlu melakukan penelurusan setiap node yang ada di dalam tree tersebut
- Pada linear struktur data seperti array, linkedlist, stack dan queue hanya ada satu cara untuk melakukan penelusuran
- Sementara pada tree, penelusuran dapat dilakukan dengan beberapa cara

Jenis Tree Traversal

- Tree traversal terbagi menjadi 3 jenis menurut order (urutan) penelusuran, yaitu
 - Inorder Traversal
 - Preorder Traversal
 - Postorder Traversal
- Ketiga jenis traversal tersebut digolongkan kedalam teknik **Depth First Traversal**
- Ada teknik lain yang digunakan untuk melakukan penelusuran yaitu **Breadth First Traversal (Level Order Traversal)**. Namun, teknik ini tidak dibahas pada materi struktur data ini

Jenis Tree Traversal



Jenis Tree Traversal

Breadth First Search (BFS)	Depth First Search (DFS)
BFS visit nodes level by level.	DFS visit nodes by depth.
BFS is slower and require more memory.	DFS is faster and require less memory.
It uses queue data structure.	It uses stack data structure.
Application: Find shortest path between 2 node, Find all connected component etc.	Application: Topological sorting, find articulation point, solving puzzle etc.

Depth First Search: Time Complexity- $O(|V|+|E|)$ where V is vertex and E is edge.
Space Complexity- $O(|V|)$

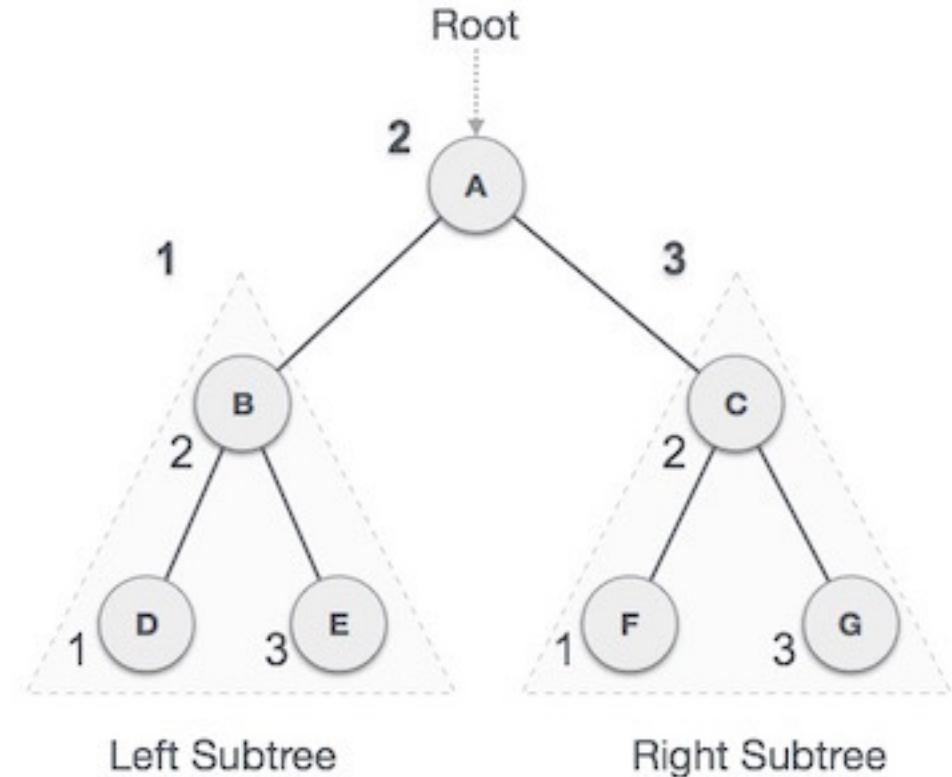
Breadth First Search: Time Complexity - $O(|V|+|E|)$ where V is vertex and E is edge.
Space Complexity- $O(|V|)$

Inorder Traversal

- Langkah-Langkah pada **inorder traversal** yaitu
 - Pertama, telusuri semua node subtree pada sebelah kiri root
 - Kemudian, penelusuran dilanjutkan ke node root
 - Terakhir, telusuri semua node subtree pada sebelah kanan root

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

- Implementasi
 - https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal_in_c.htm

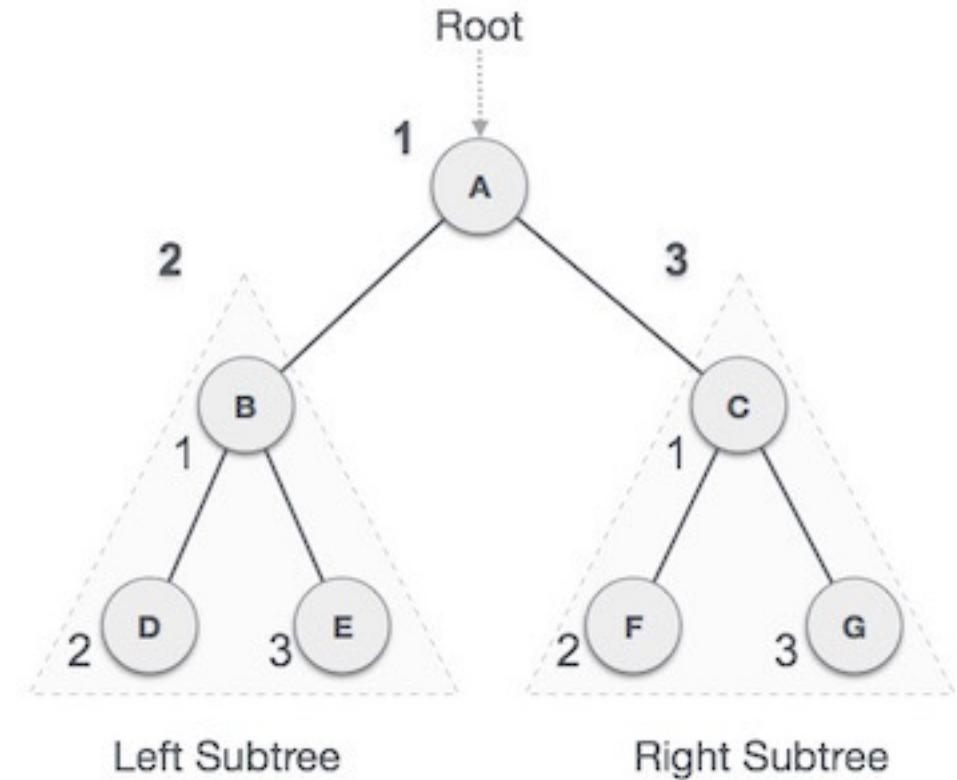


$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Preorder Traversal

- Langkah-Langkah pada **preorder traversal** yaitu
 - Pertama, kunjungi node root
 - Kemudian, telusuri semua node yang ada pada sebelah kiri subtree
 - Terakhir, telusuri semua node yang ada pada sebelah kanan subtree

```
display(root->data)  
preorder(root->left)  
preorder(root->right)
```



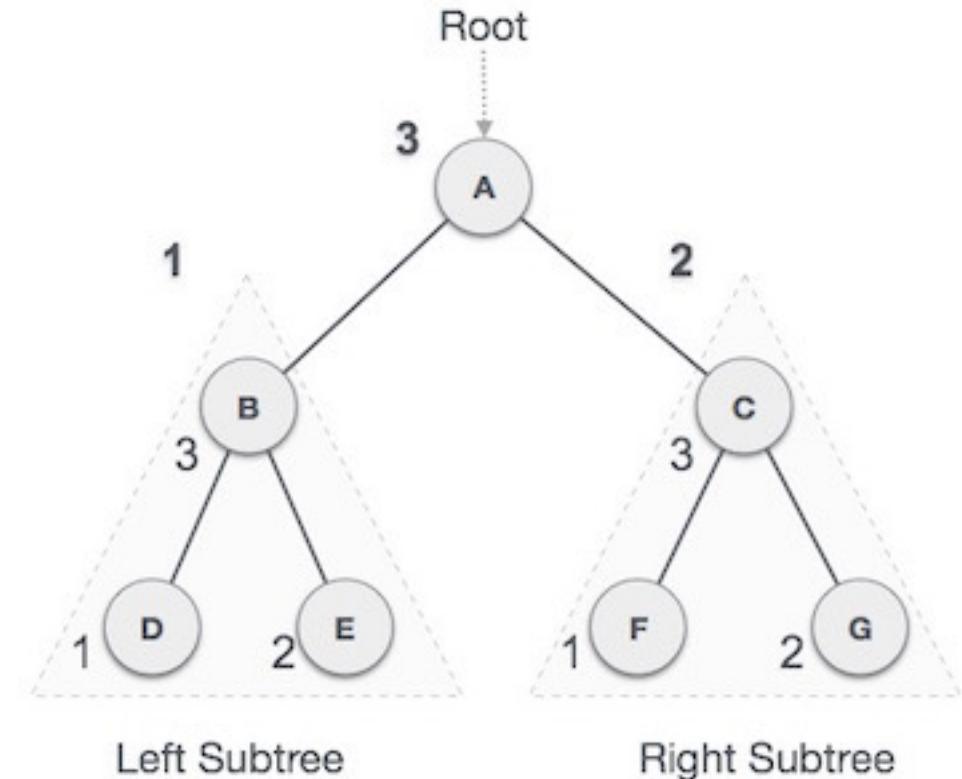
- Implementasi
 - https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal_in_c.htm

A → B → D → E → C → F → G

Postorder Traversal

- Langkah-Langkah pada **postorder traversal** yaitu
 - Pertama, telusuri semua node yang ada pada sebelah kiri subtree
 - Kemudian, telusuri semua node yang ada pada sebelah kanan subtree
 - Terakhir, telusuri node root

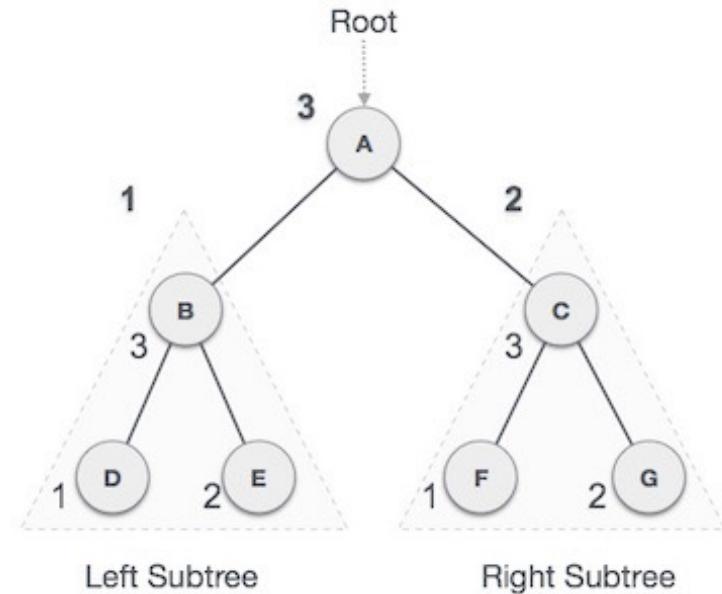
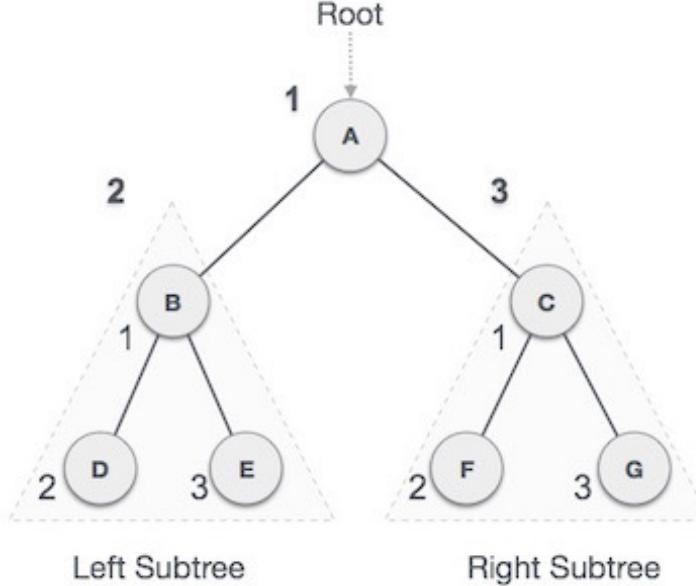
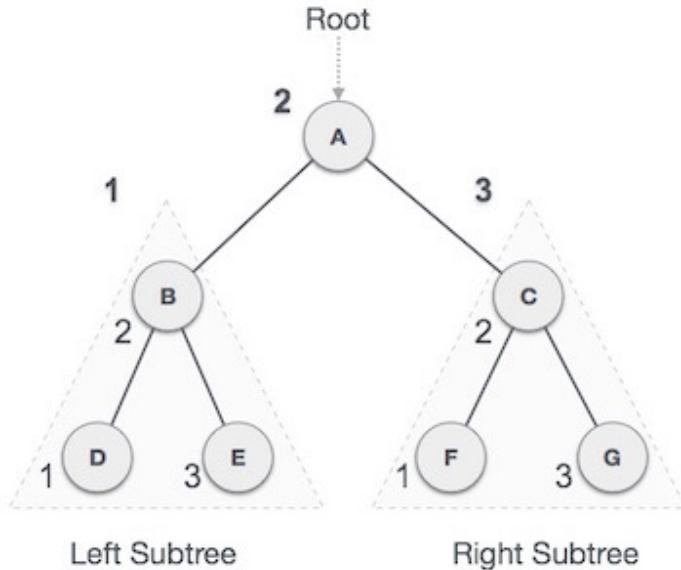
```
postorder(root->left)  
postorder(root->right)  
display(root->data)
```



- Implementasi
 - https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal_in_c.htm

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Perbandingan Tree Traversal



In-Order Traversal

$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

Pre-Order Traversal

$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

Post-Order Traversal

$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

Practice

- In-Order Tree Traversal
 - <https://practice.geeksforgeeks.org/problems/inorder-traversal/1>
- Pre-Order Tree Traversal
 - <https://practice.geeksforgeeks.org/problems/preorder-traversal/1>
- Post-Order Tree Traversal
 - <https://practice.geeksforgeeks.org/problems/postorder-traversal/1>

Q & A

Referensi

- [https://www.tutorialspoint.com/data structures algorithms/tree traversal.htm](https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm)
- <https://www.programiz.com/dsa/tree-traversal>
- <https://www.javatpoint.com/data-structure-tree-travelsal>

INF218

Struktur Data & Algoritma

Simple Sorting:
Bubble, Selection and Insertion

Alim Misbullah, S.Si., M.S.

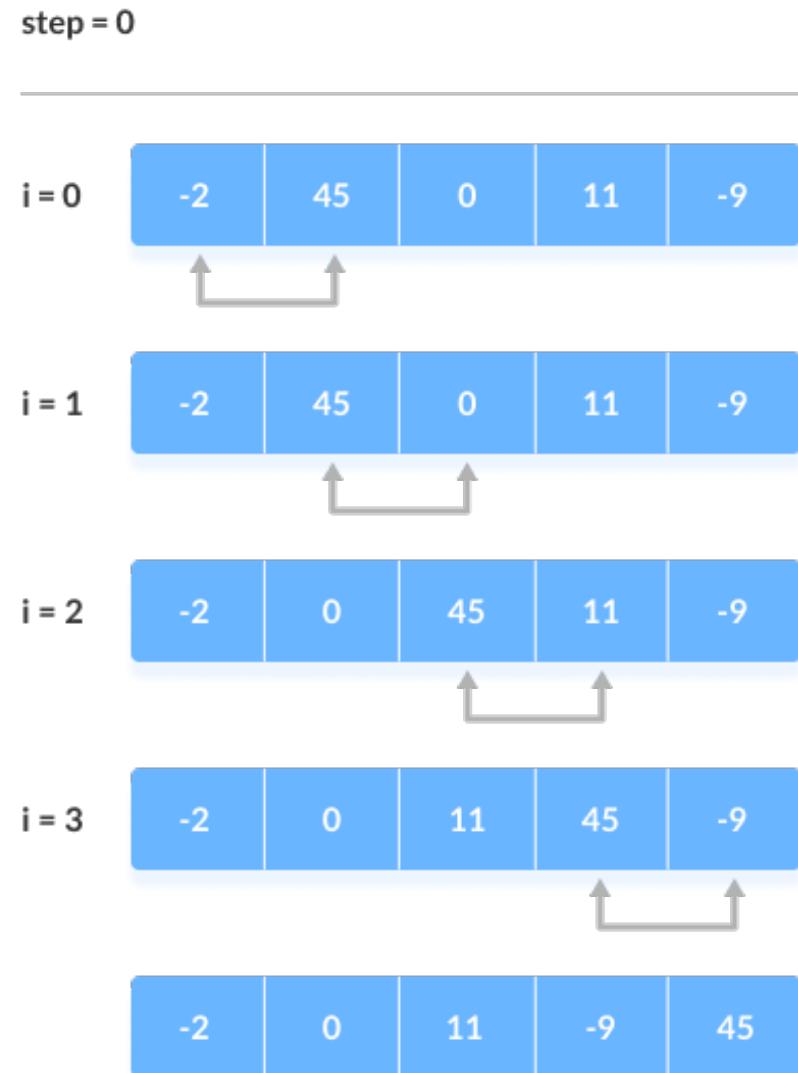
Bubble Sort

Definisi Bubble Sort

- Bubble sort adalah sebuah algoritma pengurutan yang membandingkan 2 nilai yang bersebelahan dan diganti posisinya sampai elemen tersebut terurut secara keseluruhan
- Seperti perpindahan gelembung udara di dalam air yang muncul ke permukaan
- Setiap elemen pada array akan berpindah ke bagian akhir index pada setiap iterasi
- Oleh karena itu, metode pengurutan ini disebut dengan **Bubble Sort**

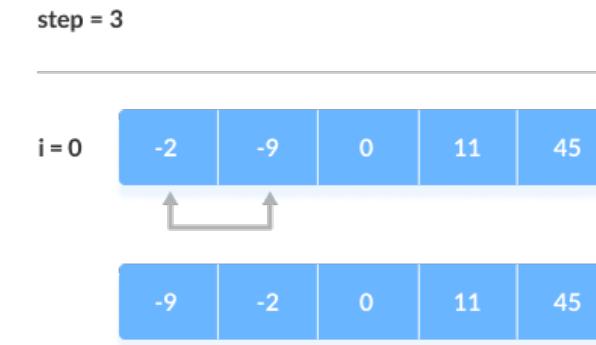
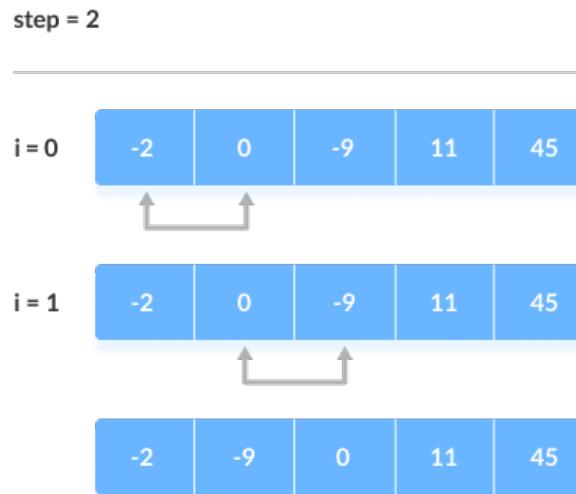
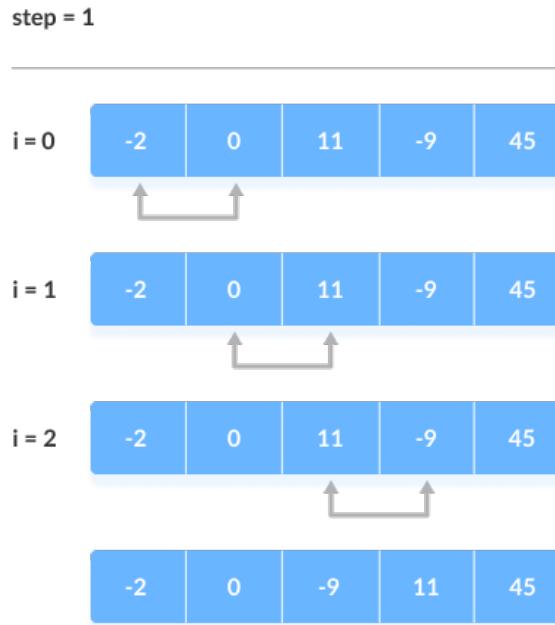
Cara Kerja Bubble Sort

- Misalnya, kita akan melakukan pengurutan data secara **ascending order**
- Iterasi Pertama (Banding dan Tukar Posisi)
 - Mulai dari index pertama, bandingkan elemen pertama dan elemen kedua
 - Jika elemen pertama lebih besar dari elemen kedua, maka tukar posisi
 - Sekarang, bandingkan elemen kedua dan elemen ketiga. Tukar posisi jika kedua elemen tersebut tidak pada posisi yang tepat urutannya
 - Proses tersebut akan dikerjakan sampai elemen terakhir



Cara Kerja Bubble Sort

- Iterasi selanjutnya



Algoritma

```
bubbleSort(array)
    for i <- 1 to indexOfLastUnsortedElement-1
        if leftElement > rightElement
            swap leftElement and rightElement
    end bubbleSort
```

Implementasi Bubble Sort

- Bahasa C
<https://www.programiz.com/dsa/bubble-sort#c-code>
- Bahasa Java
<https://www.programiz.com/dsa/bubble-sort#java-code>
- Python
<https://www.programiz.com/dsa/bubble-sort#python-code>
- C++
<https://www.programiz.com/dsa/bubble-sort#cpp-code>

Optimisasi Bubble Sort

- Pada algoritma bubble sort di atas, proses perbandingan elemen tetap akan dilakukan meskipun array sudah terurut. Hal ini akan meningkatkan waktu eksekusi
- Solusinya adalah dengan menggunakan extra variable yaitu **swapped**.
- Setelah iterasi pertama, variabel swapped tersebut akan selalu bernilai salah ketika tidak ada nilai yang diganti posisinya. Namun, jika ada nilai yang ditukar posisinya, maka variable swapped akan bernilai benar. Artinya, ada nilai di array yang belum terurut
- Jika, tidak ada perubahan maka tidak perlu dilakukan iterasi berikutnya dan array sudah terurut

Optimisasi Bubble Sort

- Algoritma Optimized Bubble Sort

```
bubbleSort(array)
    swapped <- false
    for i <- 1 to indexOfLastUnsortedElement-1
        if leftElement > rightElement
            swap leftElement and rightElement
            swapped <- true
    end bubbleSort
```

- Bahasa C
<https://www.programiz.com/dsa/bubble-sort#c-code>
- Bahasa Java
<https://www.programiz.com/dsa/bubble-sort#java-code>
- Python
<https://www.programiz.com/dsa/bubble-sort#python-code>
- C++
<https://www.programiz.com/dsa/bubble-sort#cpp-code>

Bubble Sort Complexity

Time Complexity	
Best	$O(n)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	
Stability	
	Yes

Bubble Sort Complexity

- Time Complexity
 - Worst Case Complexity
 - Jika kita ingin mengurutkan dalam bentuk ascending order dan arraynya dalam bentuk descending order, maka worst case complexity akan terjadi
 - Average Case Complexity
 - Akan terjadi ketika elemen sebuah array tersusun secara bercampur (bukan ascending atau descending)
 - Best Case Complexity
 - Jika array sudah terurut, maka tidak perlu dilakukan pengurutan kembali
- Space Complexity
 - Space complexity adalah $O(1)$ karena sebuah variable extra digunakan untuk pertukaran nilai
 - Pada Optimized Bubble Sort, dua extra variable digunakan sehingga space complexity menjadi $O(2)$

Bubble Sort Application

- Bubble Sort digunakan jika
 - Kompleksitas pengurutan bukan menjadi sebuah pertimbangan
 - Kode program yang singkat dan sederhana menjadi sebuah pertimbangan

Selection Sort

Definisi Selection Sort

- Selection Sort merupakan sebuah algoritma pengurutan yang memilih nilai terkecil dari sebuah array yang belum terurut dalam setiap iterasi dan menempatkan nilai terkecil tersebut pada bagian awal dari array yang belum terurut (**jika pengurutannya adalah ascending**)
- Algoritma selection sort akan mengatur 2 subarray di dalam sebuah array yang akan diurutkan
 - Subarray yang sudah terurut
 - Subarray yang belum terurut
- Pada setiap iterasi, elemen terkecil akan diambil dari subarray yang belum terurut dan akan dipindahkan ke bagian subarray yang sudah terurut

Definisi Selection Sort

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

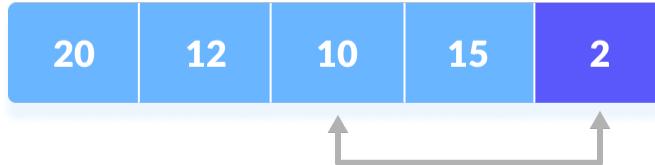
// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Cara Kerja Selection Sort

- Jadikan elemen pertama dari sebuah array sebagai **minimum**
- Bandingkan **minimum** dengan elemen kedua. Jika elemen kedua lebih kecil dari **minimum**, maka jadikan elemen kedua sebagai **minimum**
- Bandingkan elemen ketiga dengan **minimum**. Jika elemen ketiga lebih kecil dari **minimum**, maka jadikan elemen ketiga sebagai **minimum**
- Proses tersebut akan berlangsung sampai elemen terakhir dari sebuah array
- Setelah sebuah iterasi selesai, maka elemen terkecil akan ditempatkan pada bagian awal dari array yang belum terurut

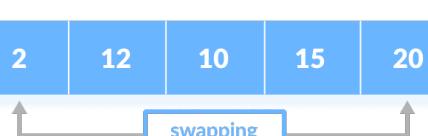
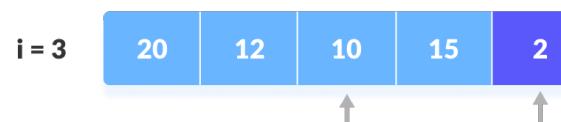
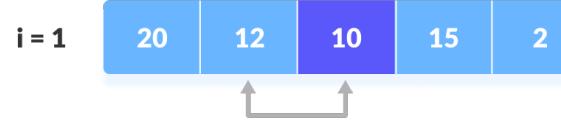
Cara Kerja Selection Sort



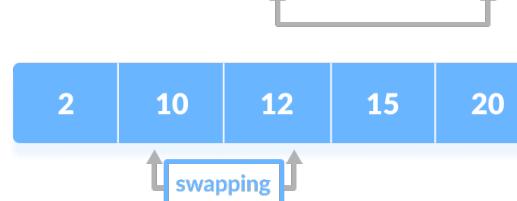
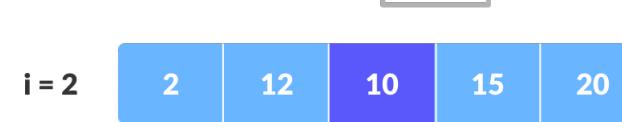
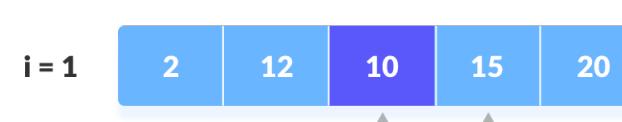
```
selectionSort(array, size)
repeat (size - 1) times
set the first unsorted element as the minimum
for each of the unsorted elements
    if element < currentMinimum
        set element as new minimum
    swap minimum with first unsorted position
end selectionSort
```

Cara Kerja Selection Sort

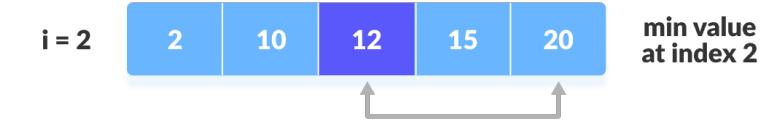
step = 0



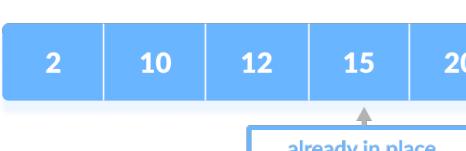
step = 1



step = 2



step = 3



Implementasi Selection Sort

- Bahasa C

<https://www.programiz.com/dsa/selection-sort#c-code>

- Bahasa Java

<https://www.programiz.com/dsa/selection-sort#java-code>

- Python

<https://www.programiz.com/dsa/selection-sort#python-code>

- C++

<https://www.programiz.com/dsa/selection-sort#cpp-code>

Selection Sort Complexity

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	
$O(1)$	
Stability	
No	

Selection Sort Complexity

- Time Complexity
 - Worst Case Complexity
 - Jika kita ingin mengurutkan dalam bentuk ascending order dan arraynya dalam bentuk descending order, maka worst case complexity akan terjadi
 - Average Case Complexity
 - Akan terjadi ketika elemen sebuah array tersusun secara bercampur (bukan ascending atau descending)
 - Best Case Complexity
 - Jika array sudah terurut, maka tidak perlu dilakukan pengurutan kembali
- Space Complexity
 - Space complexity adalah $O(1)$ karena sebuah variable extra digunakan untuk pertukaran nilai

Insertion Sort

Definisi Insertion Sort

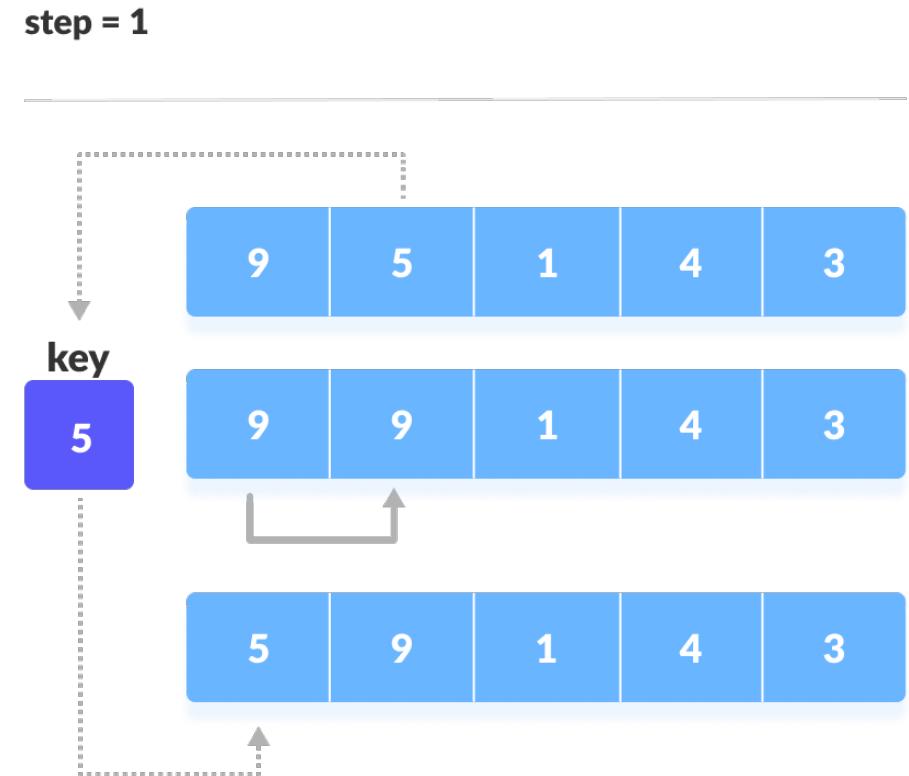
- Insertion sort adalah sebuah algoritma pengurutan yang menempatkan sebuah elemen yang belum terurut pada tempat yang sesuai di setiap iterasi
- Insertion sort bekerja seperti mengurutkan kartu di tangan pada sebuah permainan kartu
- Kita mengasumsikan bahwa kartu pertama sudah terurut, kemudian kita memilih sebuah kartu yang belum terurut. Jika yang dipilih tersebut lebih besar dari kartu yang sudah terurut, maka kita tempatkan disebelah kanan (pada kasus ascending). Namun, jika lebih kecil maka kartu tersebut ditempatkan diposisi sebelah kiri kartu yang sudah terurut
- Cara yang sama juga digunakan pada algoritma **Insertion Sort**

Cara Kerja Insertion Sort

- Misalnya kita ingin mengurutkan array berikut



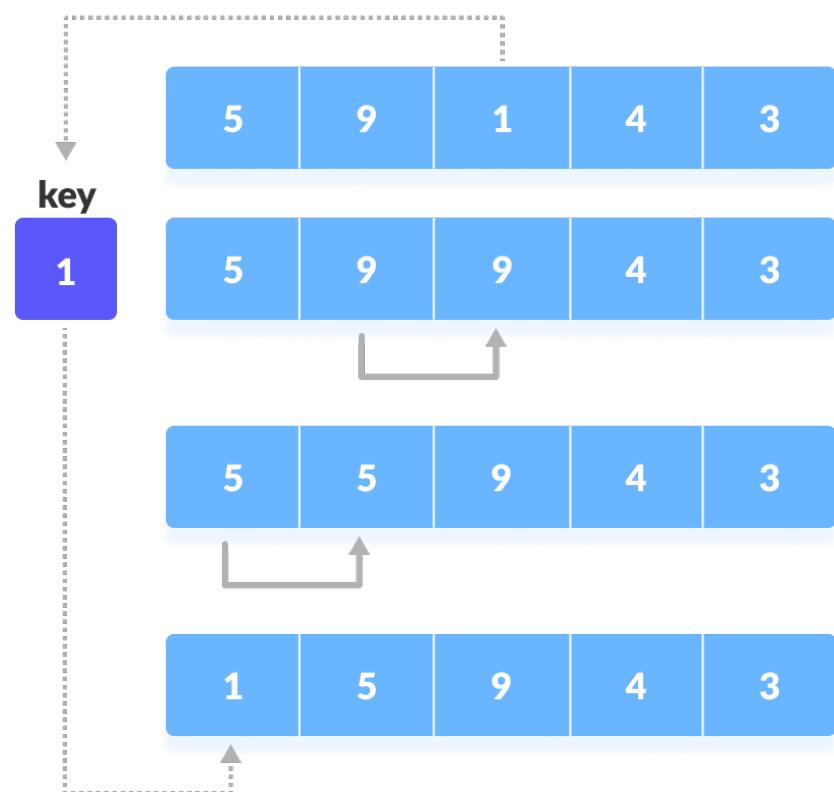
- Elemen pertama di dalam array diasumsikan sudah terurut
- Ambil elemen kedua dan tempatkan secara terpisah pada sebuah variable **key**
- Bandingkan **key** dengan elemen pertama. Jika elemen pertama lebih besar dari **key**, maka **key** ditempatkan tepat di depan elemen pertama



Cara Kerja Insertion Sort

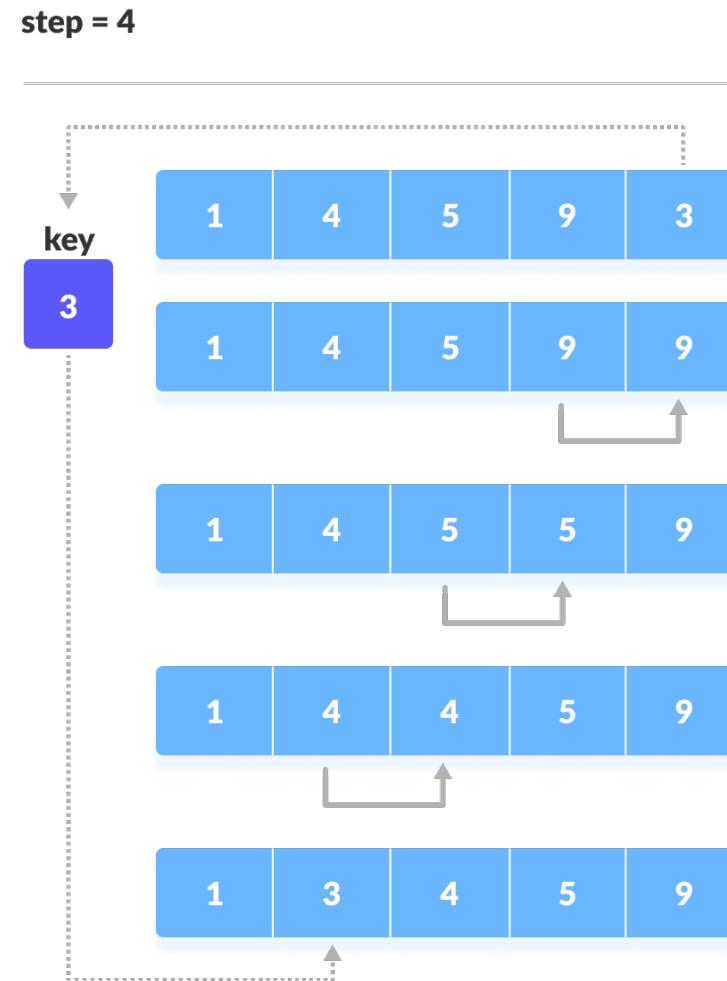
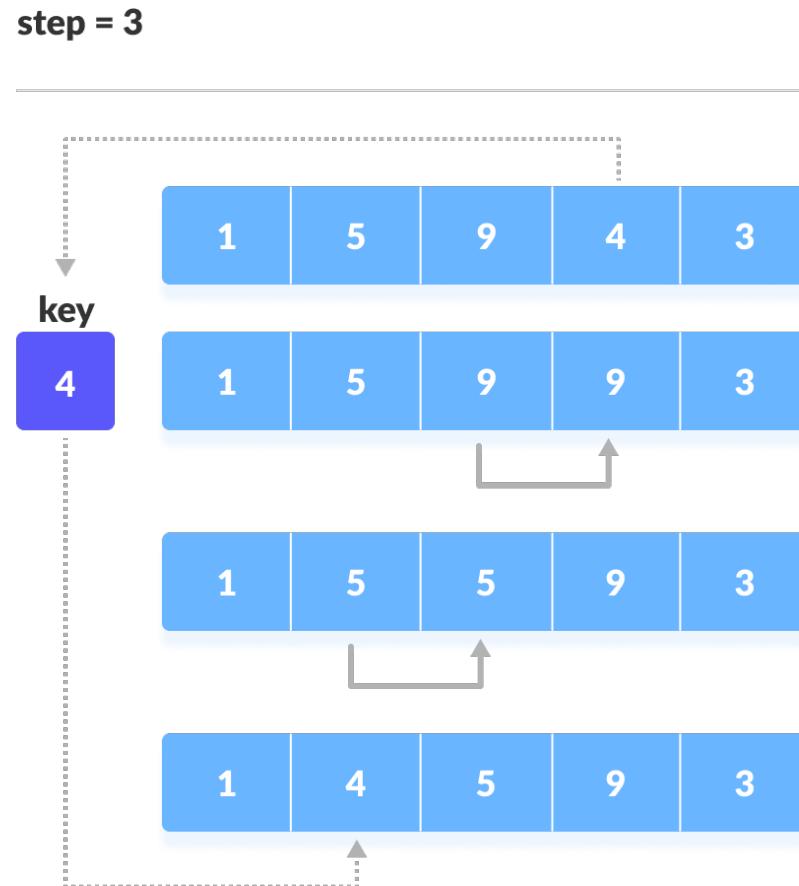
- Sekarang, kedua elemen sudah terurut
- Selanjutnya, bandingkan elemen ketiga dengan elemen sebelah kirinya. Tempatkan elemen yang lebih kecil dibagian kirinya
- Jika tidak ada elemen yang lebih kecil, maka tempatkan elemen tersebut di index pertama array

step = 2



Cara Kerja Insertion Sort

- Seperti pada langkah sebelumnya, tempatkan elemen yang belum terurut pada posisi yang sesuai



Algoritma Insertion Sort

```
insertionSort(array)
    mark first element as sorted
    for each unsorted element X
        'extract' the element X
        for j <- lastSortedIndex down to 0
            if current element j > X
                move sorted element to the right by 1
            break loop and insert X here
    end insertionSort
```

Implementasi Selection Sort

- Bahasa C
<https://www.programiz.com/dsa/insertion-sort#c-code>
- Bahasa Java
<https://www.programiz.com/dsa/insertion-sort#java-code>
- Python
<https://www.programiz.com/dsa/insertion-sort#python-code>
- C++
<https://www.programiz.com/dsa/insertion-sort#cpp-code>

Insertion Sort Complexity

Time Complexity	
Best	$O(n)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	
$O(1)$	
Stability	
Yes	

Insertion Sort Complexity

- Time Complexity
 - Worst Case Complexity
 - Jika kita ingin mengurutkan dalam bentuk ascending order dan arraynya dalam bentuk descending order, maka worst case complexity akan terjadi
 - Average Case Complexity
 - Akan terjadi ketika elemen sebuah array tersusun secara bercampur (bukan ascending atau descending)
 - Best Case Complexity
 - Jika array sudah terurut, maka tidak perlu dilakukan pengurutan kembali
- Space Complexity
 - Space complexity adalah $O(1)$ karena sebuah variable extra digunakan untuk pertukaran nilai

Q & A

Referensi

- <https://www.programiz.com/dsa/bubble-sort>
- <https://www.programiz.com/dsa/selection-sort>
- <https://www.programiz.com/dsa/insertion-sort>