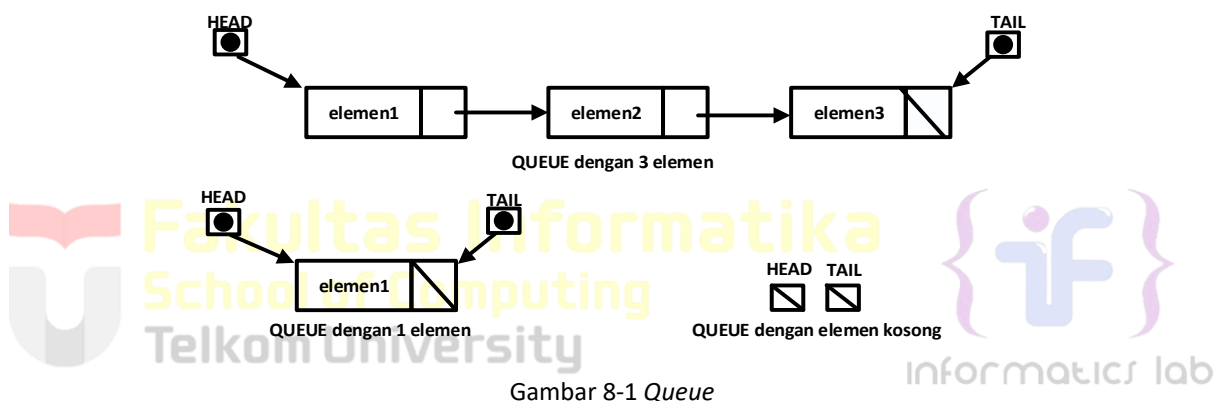


Modul 8 QUEUE

8.1 Pengertian Queue

Queue (dibaca : kyu) merupakan struktur data yang dapat diumpamakan seperti sebuah antrean. Misalkan antrean pada loket pembelian tiket Kereta Api. Orang yang akan mendapatkan pelayanan yang pertama adalah orang pertamakali masuk dalam antrean tersebut dan yang terakhir masuk dia akan mendapatkan layanan yang terakhir pula. Jadi prinsip dasar dalam *Queue* adalah **FIFO** (*First in Fisrt out*), proses yang pertama masuk akan diakses terlebih dahulu. Dalam pengimplementasian struktur *Queue* dalam C dapat menggunakan tipe data *array* dan *linked list*.

Dalam praktikum ini hanya akan dibahas pengimplementasian *Queue* dalam bentuk *linked list*. Implementasi *Queue* dalam *linked list* sebenarnya tidak jauh berbeda dengan operasi *list* biasa, malahan lebih sederhana. Karena sesuai dengan sifat FIFO dimana proses *delete* hanya dilakukan pada bagian **Head** (depan *list*) dan proses *insert* selalu dilakukan pada bagian **Tail** (belakang *list*) atau sebaliknya, tergantung dari persepsi masing-masing. Dalam penerapannya *Queue* dapat diterapkan dalam *Singly linked list* dan *Doubly linked list*.



Contoh pendeklarasian struktur data *queue*:

```
1  #ifndef queue_H
2  #define queue_H
3  #define Nil NULL
4
5  typedef int infotype; /*tipe data dalam queue */
6  typedef struct elmQueue *address; /* tipe data pointer untuk elemen queue */
7  struct elmQueue{
8      infotype info;
9      address next;
10 }; /*tipe data elemen queue */
11 /* pendeklarasian tipe data queue */
12 struct queue {
13     address head, tail;
14 };
15 #endif
16
17
18
19
```

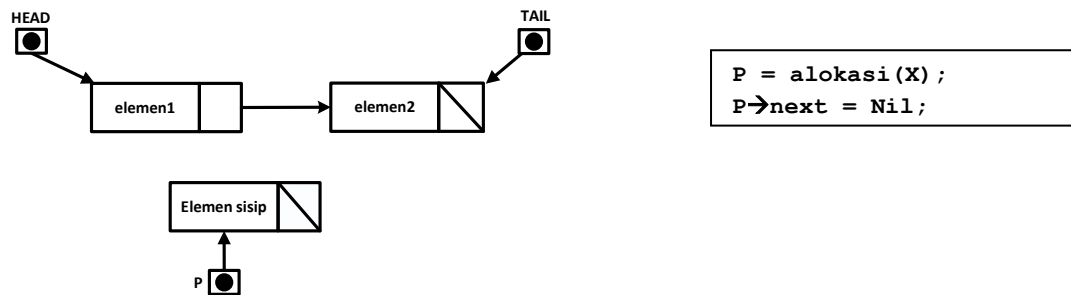
8.2 Operasi-Operasi dalam Queue

Dalam *queue* ada dua operasi utama, yaitu operasi penyisipan (*Insert/Enqueue*) dan operasi pengambilan (*Delete/Dequeue*).

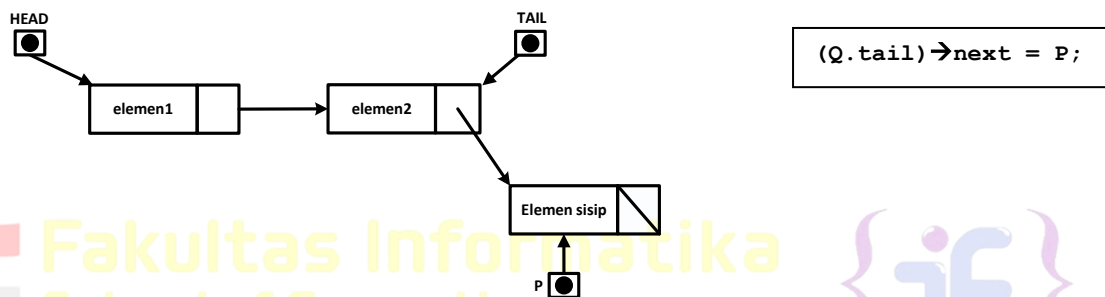
8.2.1 Insert (Enqueue)

Operasi penyisipan selalu dilakukan pada akhir (*tail*).

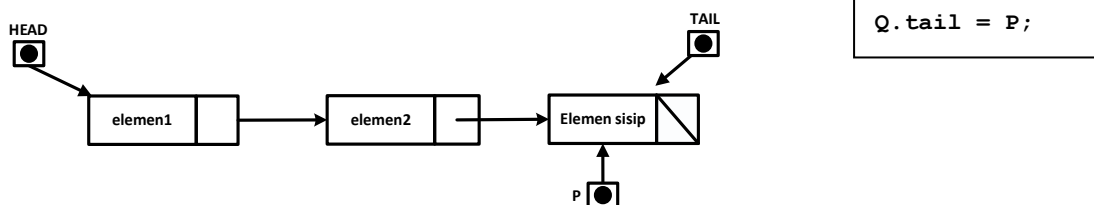
Langkah – langkah dalam proses Enqueue:



Gambar 8-2 Queue Insert 1



Gambar 8-3 Queue Insert 2

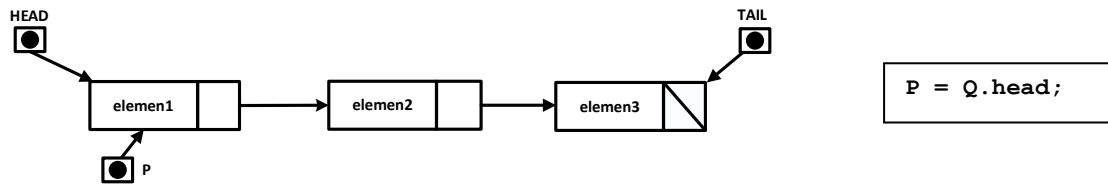


Gambar 8-4 Queue Insert 3

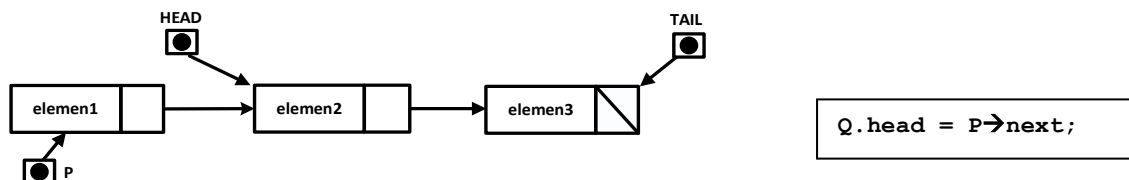
```
1  /* buat dahulu elemen yang akan disisipkan */  
2  address createElm(int X){  
3      address P = alokasi(X);  
4      P→next = Nil;  
5      return P;  
6  }  
7  
8  /* contoh syntax queue insert */  
9  void queue(address P){  
10     (Q.tail)→next = P;  
11     Q.tail = P;  
12 }
```

8.2.2 Delete (Dequeue)

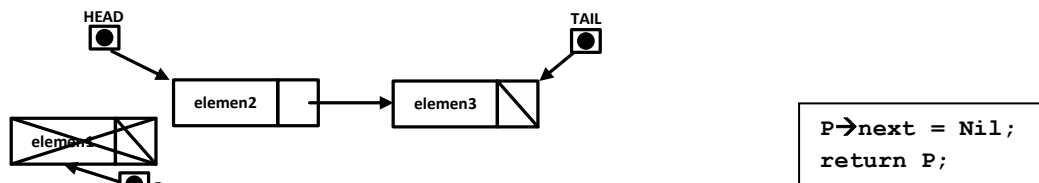
Operasi *delete* dilakukan pada awal (*head*).



Gambar 8-5 Queue Delete 1



Gambar 8-6 Queue Delete 2



Gambar 8-7 Queue Delete 3

```

1  /*contoh syntax dequeue */
2  address dequeue(address P) {
3      P = Q.head;
4      Q.head = P->next;
5      P->next = Nil;
6      return P;
7  }
  
```

8.3 Primitif-Primitif dalam Queue

Primitif-primitif pada *queue* tersimpan pada ADT *queue*, seperti pada materi sebelumnya, primitif-primitifnya tersimpan pada file *.h dan *.c.

File *.h untuk ADT *queue*:

```

1  /*file : queue .h
2  contoh ADT queue dengan representasi fisik pointer
3  representasi address dengan pointer
4  info tipe adalah integer */
5
6  #ifndef QUEUE_H_INCLUDE
7  #define QUEUE_H_INCLUDE
8
9  #define Nil NULL
10
11 typedef int infotype;
12 typedef struct elmQ *address;
13
14 struct elmQ{
15     infotype info;
16     address next;
17 };
  
```

```

18 /* deklarasi tipe data queue, terdiri dari head dan tail, queue kosong jika
19     head = Nil */
20 struct queue {
21     address head, tail;
22 };
23
24 /*prototype*/
25 /***** pengecekan apakah queue kosong *****/
26 boolean isEmpty(queue Q);
27 /*mengembalikan nilai true jika queue kosong*/
28
29 /***** pembuatan queue kosong *****/
30 void CreateQueue(queue &Q);
31 /* I.S. sembarang
32     F.S. terbentuk queue kosong*/
33
34
35 /***** manajemen memori *****/
36 void alokasi(infotype X);
37 /* mengirimkan address dari alokasi elemen
38     jika alokasi berhasil, maka nilai address tidak Nil dan jika gagal, nilai
39     address Nil */
40
41 void dealokasi(address P);
42 /* I.S. P terdefinisi
43     F.S. memori yang digunakan P dikembalikan ke sistem */
44
45 /***** pencarian sebuah elemen list *****/
46 address findElm(queue Q, infotype X);
47 /* mencari apakah ada elemen queue dengan P→info = X
48     jika ada, mengembalikan address elemen tsb, dan Nil jika sebaliknya */
49
50 boolean fFindElm(queue Q address P);
51 /* mencari apakah ada elemen queue dengan alamat P
52     mengembalikan true jika ada dan false jika tidak ada */
53
54 /***** penambahan elemen *****/
55 void insert(queue &Q, address P);
56 /* I.S. sembarang, P sudah dialokasikan
57     F.S. menempatkan elemen beralamat P pada akhir queue*/
58
59 /***** penghapusan sebuah elemen *****/
60 void delete(queue &Q, address &P);
61 /* I.S. queue tidak kosong
62     F.S. menunjuk elemen pertama queue, head dari queue menunjuk pada next
63     elemen head yang lama. Queue mungkin menjadi kosong */
64
65 /***** proses semua elemen queue *****/
66 void printInfo(queue Q);
67 /* I.S. queue mungkin tidak kosong
68     F.S. jika queue tidak kosong, maka menampilkan semua info yang ada pada
69     queue */
70
71 void nbList(queue Q);
72 /* mengembalikan jumlah elemen pada queue */
73
74 void delAll(queue &Q);
75 /* menghapus semua elemen pada queue dan semua elemen di-dealokasi */
76
77 #endif
78
79
80
81

```

8.4 Queue (Representasi Tabel)

8.4.1 Pengertian

Pada dasarnya representasi *queue* menggunakan tabel sama halnya dengan menggunakan *pointer*. Perbedaan yang mendasar adalah pada management memori serta keterbatasan jumlah antreannya. Untuk lebih jelasnya perhatikan perbedaan representasi tabel dan *pointer* pada *queue* dibawah ini.

Tabel 8-1 Perbedaan *Queue* Representasi Table dan *Pointer*

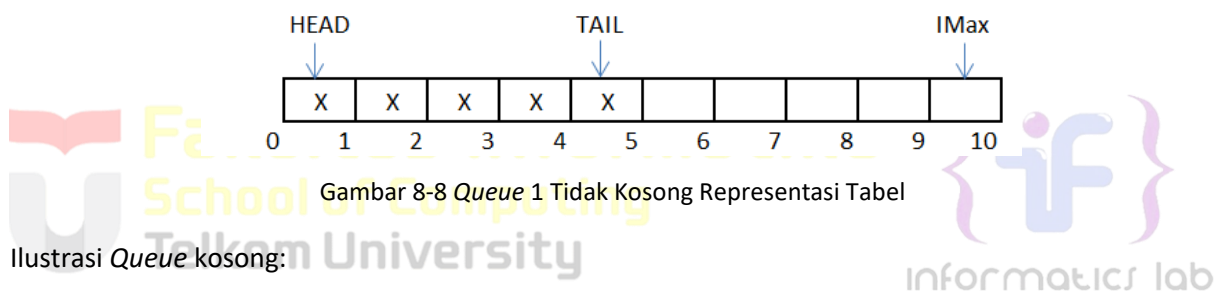
No	Representasi Table	Representasi <i>Pointer</i>
1	Jumlah <i>Queue</i> terbatas	Jumlah <i>Queue</i> tak-terbatas
2	Tidak ada management memori	ada management memori

Ada beberapa macam-macam bentuk *queue*, yaitu:

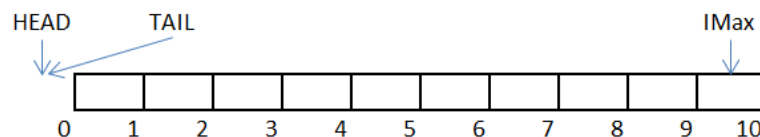
A. Alternatif 1

Tabel dengan hanya representasi *TAIL* adalah indeks elemen terakhir, *HEAD* selalu di-set sama dengan 1 jika *Queue* tidak kosong. Jika *Queue* kosong, maka *HEAD*=0.

Ilustrasi *Queue* tidak kosong, dengan 5 elemen :



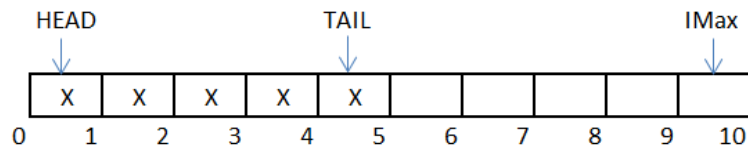
Ilustrasi *Queue* kosong:



Algoritma paling sederhana untuk penambahan elemen jika masih ada tempat adalah dengan “memajukan” *TAIL*. Kasus khusus untuk *Queue* kosong karena *HEAD* harus diset nilainya menjadi 1. Algoritma paling sederhana dan “naif” untuk penghapusan elemen jika *Queue* tidak kosong: ambil nilai elemen *HEAD*, geser semua elemen mulai dari *HEAD*+1 s/d *TAIL* (jika ada), kemudian *TAIL* “mundur”. Kasus khusus untuk *Queue* dengan keadaan awal berelemen 1, yaitu menyesuaikan *HEAD* dan *TAIL* dengan DEFINISI. Algoritma ini mencerminkan pergeseran orang yang sedang mengantre di dunia nyata, tapi tidak efisien.

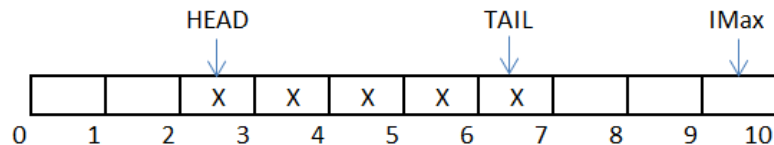
B. Alternatif 2

Tabel dengan representasi *HEAD* dan *TAIL*, *HEAD* “bergerak” ketika sebuah elemen dihapus jika *Queue* tidak kosong. Jika *Queue* kosong, maka *HEAD*=0. Ilustrasi *Queue* tidak kosong, dengan 5 elemen, kemungkinan pertama *HEAD* “sedang berada di posisi awal:



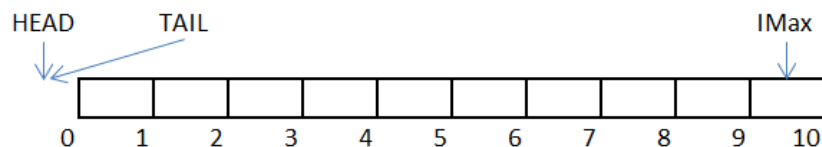
Gambar 8-10 Queue 2 Tidak Kosong 1 Representasi Table

Ilustrasi *Queue* tidak kosong, dengan 5 elemen, kemungkinan pertama *HEAD* tidak berada di posisi awal. Hal ini terjadi akibat algoritma penghapusan yang dilakukan (lihat keterangan).



Gambar 8-11 Queue 2 Tidak Kosong 2 Representasi Tabel

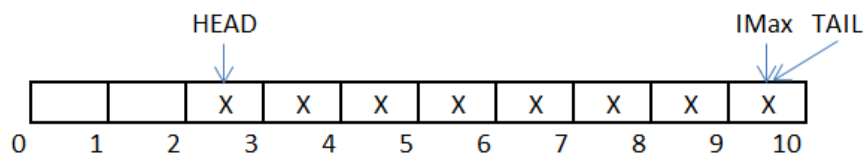
Ilustrasi *Queue* kosong:



Gambar 8-12 Queue 2 Kosong Representasi Tabel

Algoritma untuk penambahan elemen sama dengan alternatif I: jika masih ada tempat adalah dengan “memajukan” *TAIL*. Kasus khusus untuk *Queue* kosong karena *HEAD* harus diset nilainya menjadi 1. Algoritmanya sama dengan alternatif I. Algoritma untuk penghapusan elemen jika *Queue* tidak kosong: ambil nilai elemen *HEAD*, kemudian *HEAD* “maju”. Kasus khusus untuk *Queue* dengan keadaan awal berelemen 1, yaitu menyesuaikan *HEAD* dan *TAIL* dengan DEFINISI.

Algoritma ini TIDAK mencerminkan pergeseran orang yang sedang mengantre di dunia nyata, tapi efisien. Namun suatu saat terjadi keadaan *Queue* penuh tetapi “semu” sebagai berikut :

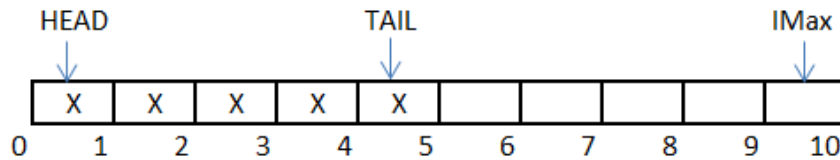


Gambar 8-13 Queue 2 Penuh “semu”

Jika keadaan ini terjadi, haruslah dilakukan aksi menggeser elemen untuk menciptakan ruangan kosong. Pergeseran hanya dilakukan jika dan hanya jika *TAIL* sudah mencapai IndexMax.

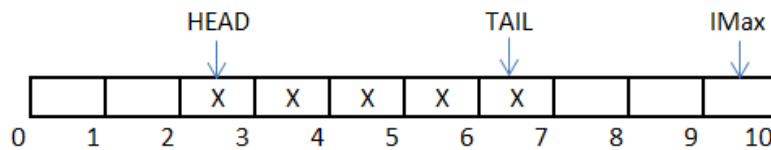
C. Alternatif 3

Tabel dengan representasi *HEAD* dan *TAIL* yang “berputar” mengelilingi indeks tabel dari awal sampai akhir, kemudian kembali ke awal. Jika *Queue* kosong, maka *HEAD*=0. Representasi ini memungkinkan tidak perlu lagi ada pergeseran yang harus dilakukan seperti pada alternatif II pada saat penambahan elemen. Ilustrasi *Queue* tidak kosong, dengan 5 elemen, dengan *HEAD* “sedang” berada di posisi awal:



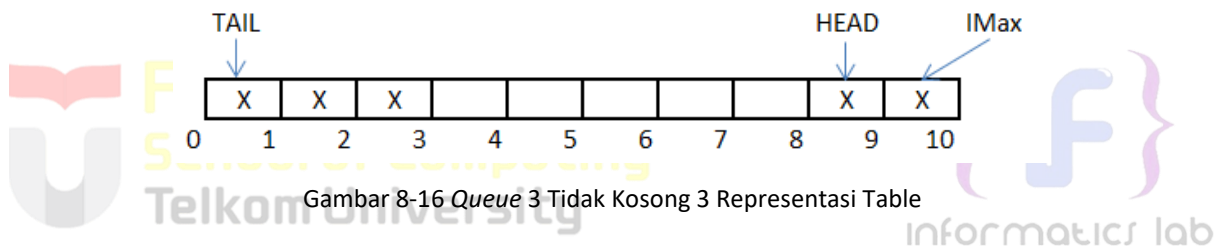
Gambar 8-14 Queue 3 Tidak Kosong 1 Representasi Table

Ilustrasi *Queue* tidak kosong, dengan 5 elemen, dengan *HEAD* tidak berada diposisi awal, tetapi masih “lebih kecil” atau “sebelum” *TAIL*. Hal ini terjadi akibat algoritma penghapusan/Penambahan yang dilakukan (lihat keterangan).



Gambar 8-15 Queue 3 Tidak Kosong 2 Representasi Table

Ilustrasi *Queue* tidak kosong, dengan 5 elemen, *HEAD* tidak berada di posisi awal, tetapi “lebih besar” atau “sesudah” *TAIL*. Hal ini terjadi akibat algoritma penghapusan/Penambahan yang dilakukan (lihat keterangan)



Gambar 8-16 Queue 3 Tidak Kosong 3 Representasi Table

Algoritma untuk penambahan elemen: jika masih ada tempat adalah dengan “memajukan” *TAIL*. Tapi jika *TAIL* sudah mencapai *IdxMax*, maka *successor* dari *IdxMax* adalah 1 sehingga *TAIL* yang baru adalah 1. Jika *TAIL* belum mencapai *IdxMax*, maka algoritma penambahan elemen sama dengan alternatif II. Kasus khusus untuk *Queue* kosong karena *HEAD* harus diset nilainya menjadi 1.

Algoritma untuk penghapusan elemen jika *Queue* tidak kosong: ambil nilai elemen *HEAD*, kemudian *HEAD* “maju”. Penentuan suatu *successor* dari indeks yang diubah/”maju” dibuat Seperti pada algoritma penambahan elemen: jika *HEAD* mencapai *IdxMAX*, maka *successor* dari *HEAD* adalah 1. Kasus khusus untuk *Queue* dengan keadaan awal berelemen 1, yaitu menyesuaikan *HEAD* dan *TAIL* dengan DEFINISI. Algoritma ini efisien karena tidak perlu pergeseran, dan seringkali strategi pemakaian tabel semacam ini disebut sebagai “circular buffer”, dimana tabel penyimpan elemen dianggap sebagai “buffer”.

Salah satu variasi dari representasi pada alternatif III adalah : menggantikan representasi *TAIL* dengan COUNT, yaitu banyaknya elemen *Queue*. Dengan representasi ini, banyaknya elemen diketahui secara eksplisit, tetapi untuk melakukan penambahan elemen harus dilakukan kalkulasi *TAIL*. Buatlah sebagai latihan.

Contoh Kasus Alternatif 2

Antrean kosong (*Head*=0, *Tail*=0)



Setelah penambahan elemen 8,3,6,9,15,1 (Head=1, Tail=6)

8	3	6	9	15	1				
1	2	3	4	5	6	7	8	9	10

Setelah penghapusan sebuah elemen (Head=2, Tail=6)

	3	6	9	15	1				
1	2	3	4	5	6	7	8	9	10

Algoritma	C++
<p>Kamus Umum</p> <p><u>constant</u> NMax: <u>integer</u> = 100 <u>constant</u> Nil: <u>integer</u> = 0 <u>type</u> infotype = <u>integer</u> <u>type</u> Queue = <TabQ: <u>array</u>[1..NMax] <u>of</u> infotype; Head, Tail: <u>integer</u>> <u>Function</u> EmptyQ(Q:Queue) → <u>boolean</u> {True jika Q antrian kosong} <u>Procedure</u> CreateEmpty(<u>output</u> Q:Queue) {I.Q. - F.Q. Terdefinisi antrian kosong Q}</p>	<pre>const int NMax=100; const int Nil=0; typedef int infotype queue { infotype TabQ[NMax]; int head, tail; } bool Empty(Queue Q){ /* True jika Q merupakan antrian kosong */ } void CreateEmpty(){ /* membuat antrian kosong */ } bool isFull(Queue Q){ /* True jika Q penuh */ if(Q.head=1 && Q.tail=Nmax){ return true; } }</pre>
<p><u>function</u> IsFullQ(Q:Queue) → <u>boolean</u> {True Q penuh} kamus: algoritma → (Q.Head = 1) and (Q.Tail=NMax)</p>	<pre>void addQ(Queue Q, infotype x){ int i; if (isFull(Q)){ cout >> "Antrean Penuh"; }else{ if(Empty(Q)){ Q.head = 1; Q.tail = 1; }else{ if(Q.tail>NMax){ Q.tail++; }else{ for(i=Q.head;i<=Q.tail;i++){ Q.TabQ[i+Q.head+1] = Q.TabQ[i]; } Q.tail = Q.tail-Q.head+2; Q.head = 1; Q.TabQ[Q.tail] = x; /*menyisipkan x */ } } } }</pre>
<p><u>Procedure</u> AddQ(<u>input/output</u> Q:Queue; <u>input</u> X:infotype) (I.Q. Q antrian, terdefinisi, mungkin kosong) (F.Q. X elemen terakhir Q jika antrian tidak penuh) Kamus i:integer Algoritma <u>if</u> IsFull(Q) <u>then</u> <u>output</u> ('Antrian penuh') <u>else</u> <u>if</u> Empty(Q) <u>then</u> Q.Head ← 1 Q.Tail ← 1 <u>else</u> <u>if</u> Q.Tail<NMax <u>then</u> Q.Tail ← Q.Tail + 1 <u>else</u> <u>for</u> i←Q.Head <u>to</u> Q.Tail <u>do</u> Q.TabQ[i-Q.Head+1] ← Q.TabQ[i] Q.Tail ← Q.Tail-Q.Head+2 Q.Head ← 1 Q.TabQ[Q.Tail] ← X {menyisipkan X}</p>	

<pre> Procedure DelQ(input/output Q:Queue; output X:infotype) {I.Q. Q antrian, terdefinisi, tidak kosong} {F.Q. X elemen antrian yang dihapus} Kamus i:integer Algoritma X ← Q.TabQ[Q.Head] if Q.Head=Q.Tail then CreateEmpty(Q) else Q.Head ← Q.Head+1 </pre>	<pre> void DelQ(Queue Q, infotype x){ int i; x= Q.TabQ[Q.head]; if(Q.head == Q.tail){ CreateEmpty(Q); }else{ Q.head; } } </pre>
--	---

8.4.2 Primitif-Primitif dalam Queue

Berikut ini contoh program *queue.h* dalam ADT *queue* menggunakan representasi table.

```

1  /*file : queue .h
2  contoh ADT queue dengan representasi tabel*/
3
4  #ifndef QUEUE_H_INCLUDE
5  #define QUEUE_H_INCLUDE
6
7  struct infotype {
8      char id[20];
9      char nama[20];
10 };
11
12 struct Queue {
13     infotype info[3];
14     int head;
15     int tail;
16 };
17
18 /* prototype */
19 /***** pengecekan apakah Queue penuh *****/
20 int isFull(queue Q):
21 /* mengembalikan nilai 0 jika queue penuh */
22
23 /***** pengecekan apakah Queue kosong *****/
24 int isEmpty(queue Q);
25 /* mengembalikan nilai 0 jika queue kosong */
26
27 /***** pembuatan queue *****/
28 void CreateQueue(queue &Q);
29 /* I.S. sembarang
30    F.S. terbentuk queue dengan head = -1 dan tail = -1 */
31
32 /***** penambahan elemen pada queue *****/
33 void enqueue(queue &Q, infotype X);
34 /* I.S. queue mungkin kosong
35    F.S. menambahkan elemen pada stack dengan nilai X */
36
37 /***** penghapusan elemen pada queue *****/
38 void dequeue(queue &Q);
39 /* I.S. queue tidak kosong
40    F.S. head = head + 1 */
41
42 /***** proses semua elemen pada queue *****/
43 void viewQueue(queue Q);
44 /* I.S. queue mungkin kosong
45    F.S. jika queue tidak kosong menampilkan semua info yang ada pada queue */
46
47 #endif
48
49

```

8.5 Latihan Queue

1. Buatlah ADT *Queue* menggunakan *ARRAY* sebagai berikut di dalam file *"queue.h"*:

```
Type infotype: integer
Type Queue: <
    info : array [5] of infotype {index array dalam C++
        dimulai dari 0}
    head, tail : integer
>
procedure CreateQueue (input/output Q: Queue)
function isEmptyQueue (Q: Queue) → boolean
function isFullQueue (Q: Queue) → boolean
procedure enqueue (input/output Q: Queue, input x: infotype)
function dequeue (input/output Q: Queue) → infotype
procedure printInfo (input Q: Queue)
```

Buatlah implementasi ADT *Queue* pada file *"queue.cpp"* dengan menerapkan mekanisme *queue* Alternatif 1 (*head* diam, *tail* bergerak).

```
1  int main() {
2      cout << "Hello World" << endl;
3      Queue Q;
4      createQueue(Q);
5
6      cout<<"-----"<<endl;
7      cout<<" H - T \t | Queue info"<<endl;
8      cout<<"-----"<<endl;
9      printInfo(Q);
10     enqueue(Q,5); printInfo(Q);
11     enqueue(Q,2); printInfo(Q);
12     enqueue(Q,7); printInfo(Q);
13     dequeue(Q); printInfo(Q);
14     enqueue(Q,4); printInfo(Q);
15     dequeue(Q); printInfo(Q);
16     dequeue(Q); printInfo(Q);
17
18     return 0;
19 }
```

```
Hello world?
-----
H - T   | Queue Info
-----
-1 - -1 | empty queue
0 - 0   | 5
0 - 1   | 5 2
0 - 2   | 5 2 7
0 - 1   | 2 7
0 - 0   | 7
0 - 1   | 7 4
0 - 0   | 4
-1 - -1 | empty queue
```

Gambar 8-17 Output Queue

Gambar 8-18 main Queue

2. Buatlah implementasi ADT *Queue* pada file *"queue.cpp"* dengan menerapkan mekanisme *queue* Alternatif 2 (*head* bergerak, *tail* bergerak).
3. Buatlah implementasi ADT *Queue* pada file *"queue.cpp"* dengan menerapkan mekanisme *queue* Alternatif 3 (*head* dan *tail* berputar).