3. The Queue ADT

The Queue ADT

- A queue is a block of memory where new values are added in one end called the tail of the queue and removed from the other end called the head of the queue
- A queue is a data structure in which elements are removed in the same order they were entered.
- Also known as FIFO (first in, first out) structure

The Queue ADT

- Two operations:
  - enqueue (insert)
  - dequeue (delete)

The Queue ADT

- enqueue
  - appends a value at the tail of the queue
  - tail is updated to point to the newly appended value
- dequeue
  - deletes the value found at the head of the queue
  - updates the head pointer to point to the element next to the head of the queue

The Queue ADT

Possible Errors

- Queue Underflow
  - attempt to dequeue a value from an empty queue

- Queue Overflow
  - attempt to enqueue a value into a full queue

The Queue ADT

Implementation

- Array

- Linked list

3. The Queue ADT

3.1 Array Implementation

Queue – Array Implementation

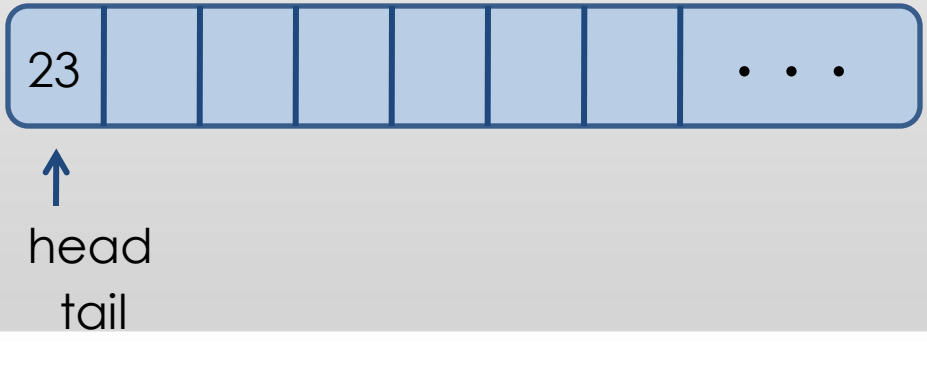
- When an array is used to implement a queue, only two locations of the array become available for access and storage.
- As elements are inserted at one end and elements are removed at the other end, the queue will naturally move from the left to the right of the array.

The Queue ADT

The Queue ADT

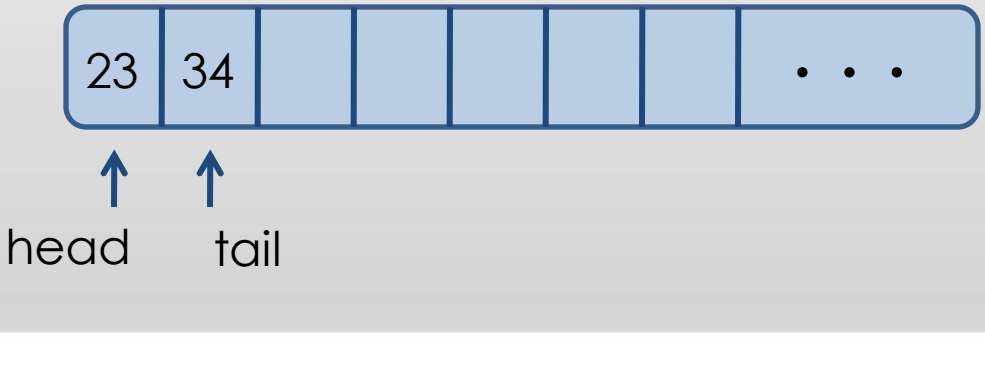
Quiz

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Quiz

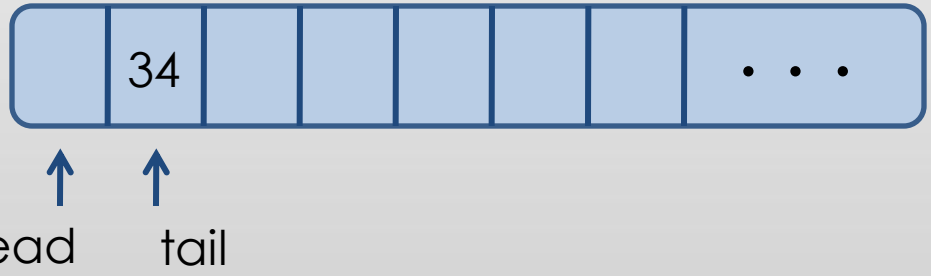
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Quiz

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

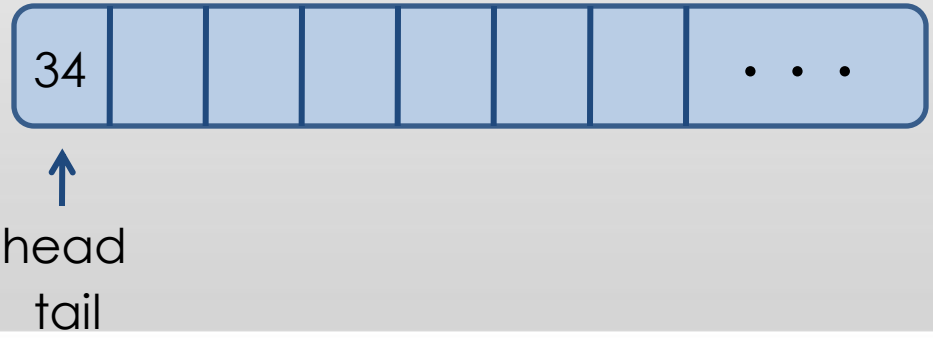
x = 23



Quiz

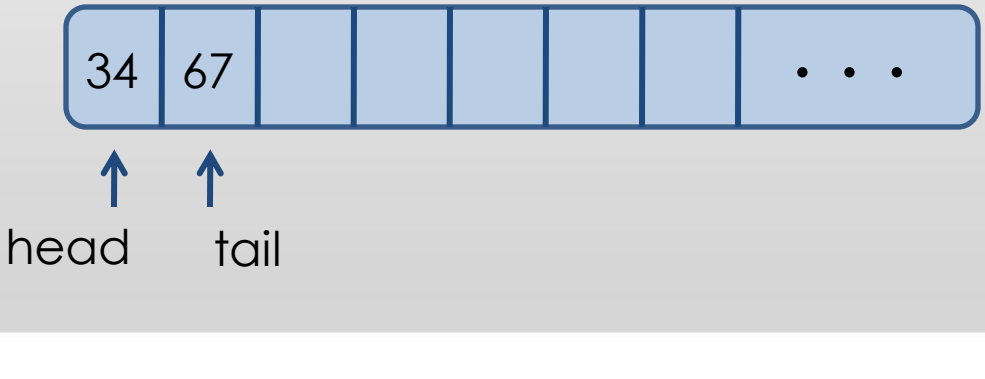
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

x = 23



Quiz

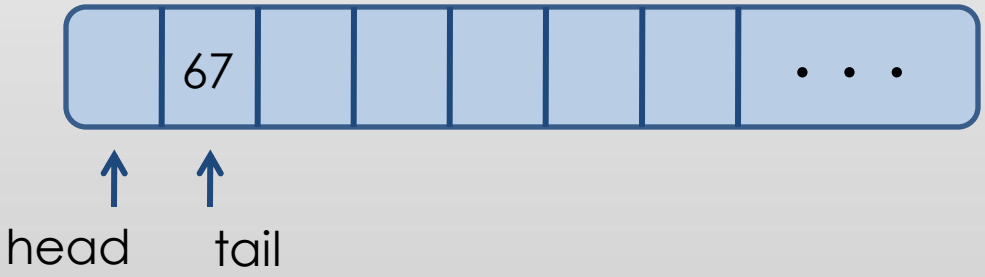
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Quiz

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

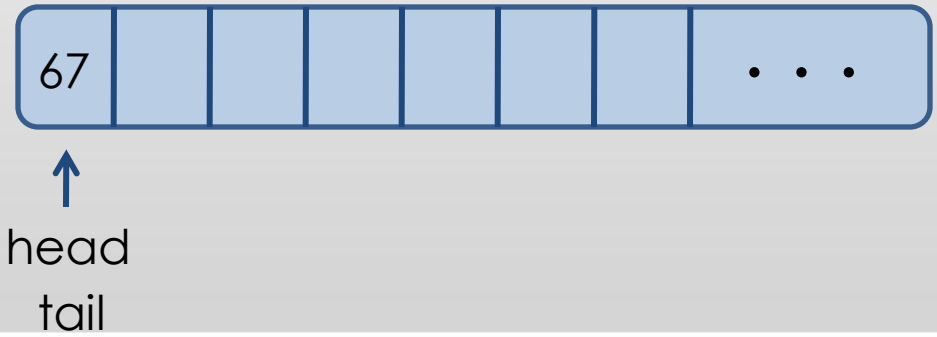
x = 34



Quiz

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

x = 34



Queue – Array Implementation

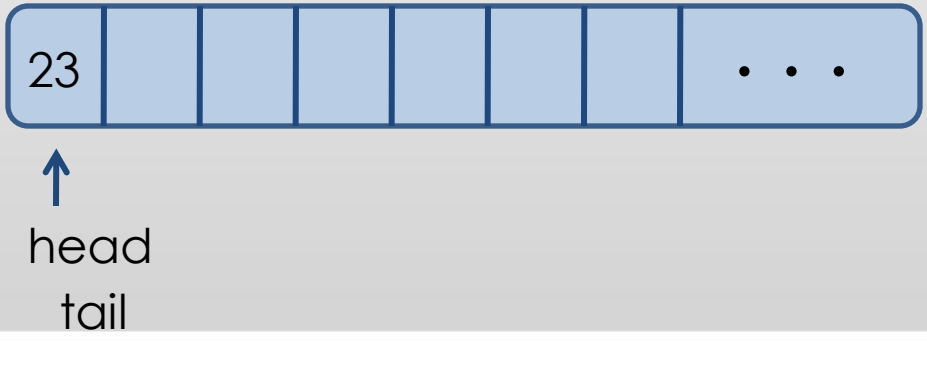
- When an array is used to implement a queue, only two locations of the array become available for access and storage.
- As elements are inserted at one end and elements are removed at the other end, the queue will naturally move from the left to the right of the array.

Queue – Array Implementation

- When the tail and eventually the head reaches the rightmost location of the array, the pointers wrap around the array.

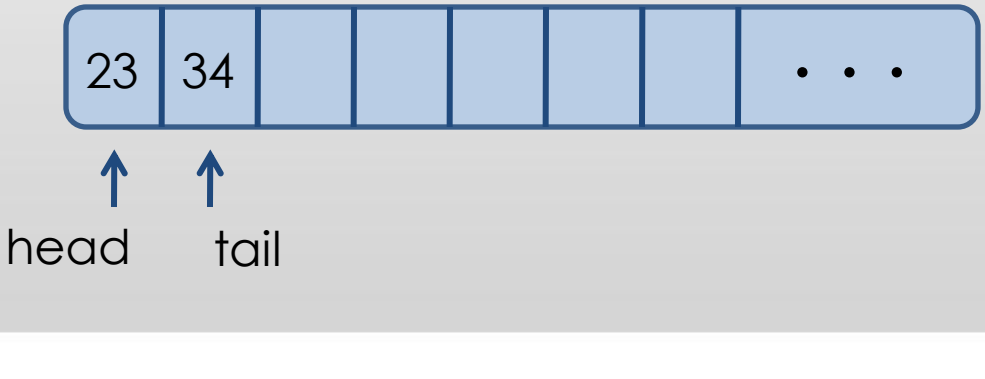
Quiz

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Quiz

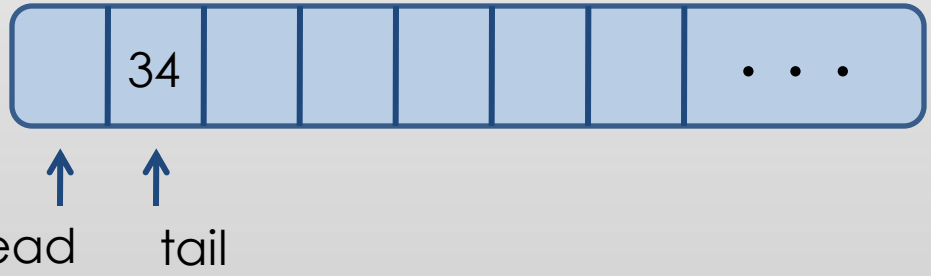
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Quiz

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

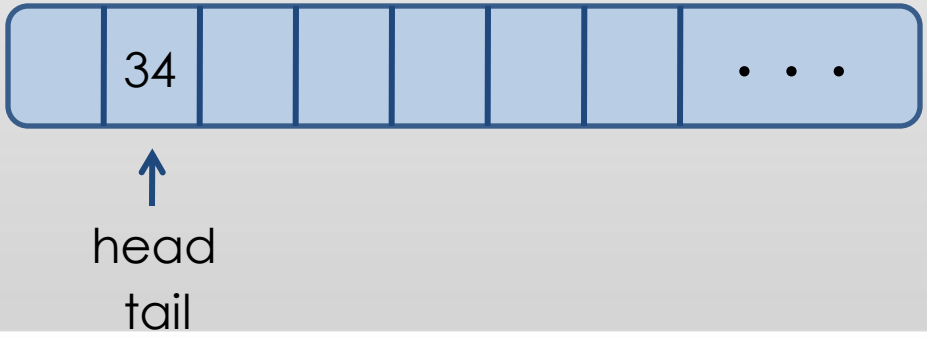
x = 23



Quiz

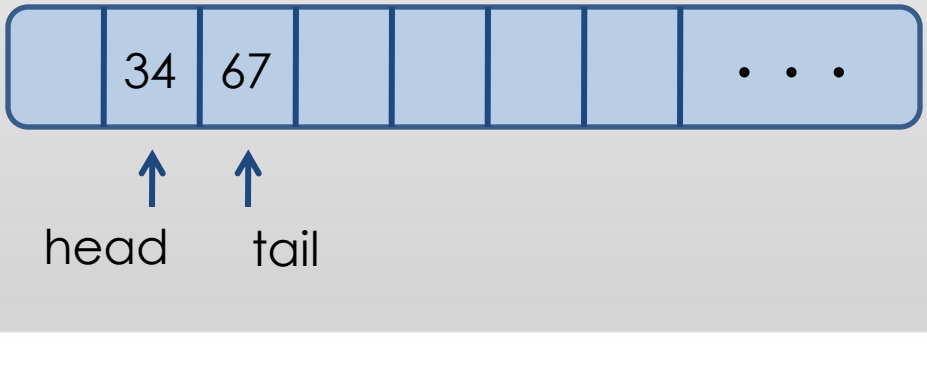
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

x = 23



Quiz

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Quiz

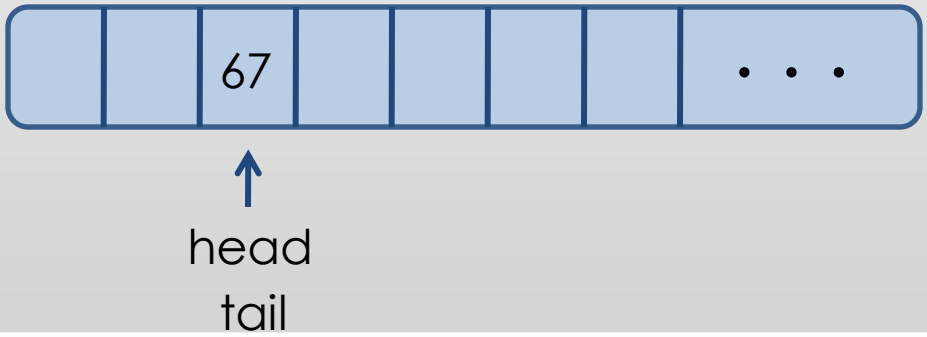
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

x = 34

Quiz

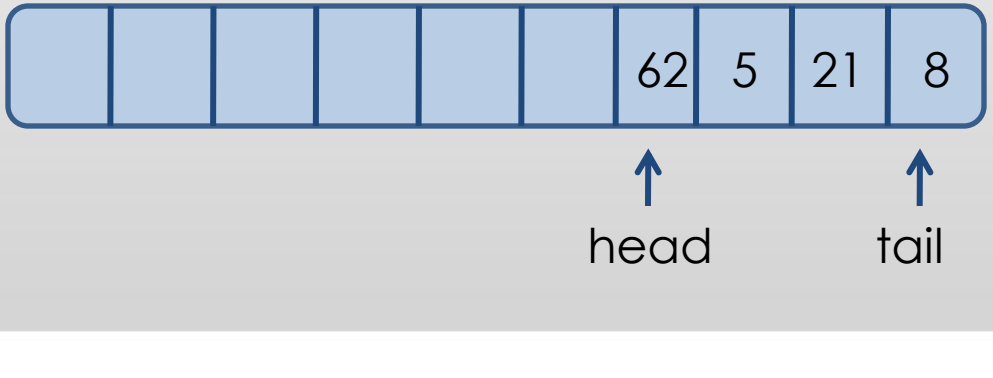
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

x = 34



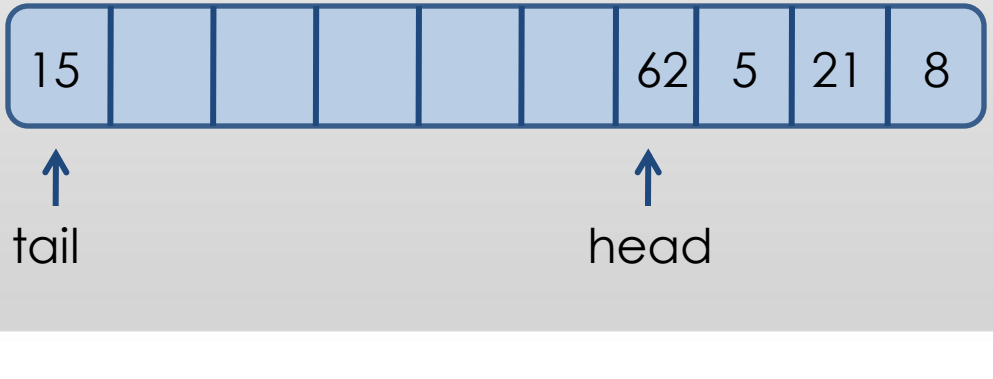
Queue – Array Implementation

```
. . .
enqueue(62);
enqueue(5);
enqueue(21);
enqueue(8);
```



Queue – Array Implementation

```
. . .
enqueue(62);
enqueue(5);
enqueue(21);
enqueue(8);
enqueue(15);
```



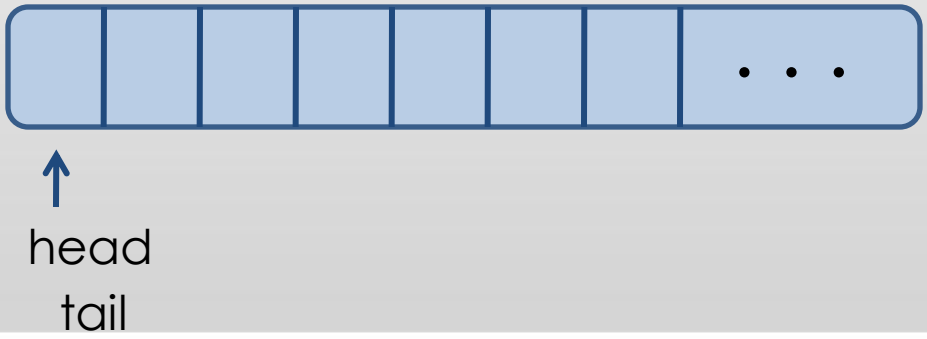
Array Implementation

```
#define LIMIT 100
int queue[LIMIT];
int tail=0,head=0;

void enqueue(int x){
  tail = (tail+1)%LIMIT;
  if (tail!=head)
    queue[tail]=x;
  else {
    printf("overflow");
    exit(1);
  }
}
```

```
int dequeue(){
  if (head!=tail){
    head = (head+1)%LIMIT;
    return(queue[head]);
  }
  else {
    printf("underflow");
    exit(1);
  }
}
```

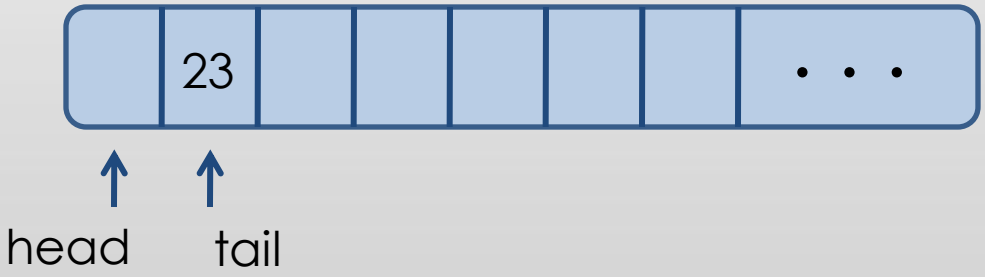
Queue – Array Implementation

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Queue – Array Implementation

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

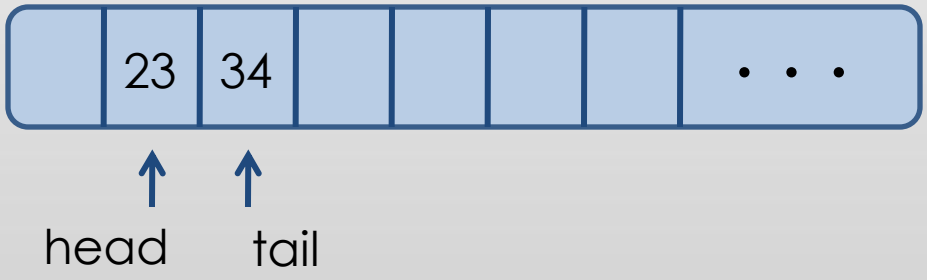


Queue – Array Implementation

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

Queue – Array Implementation

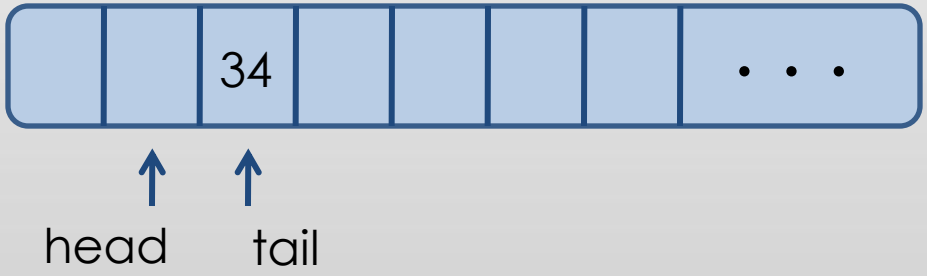
```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```



Queue – Array Implementation

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

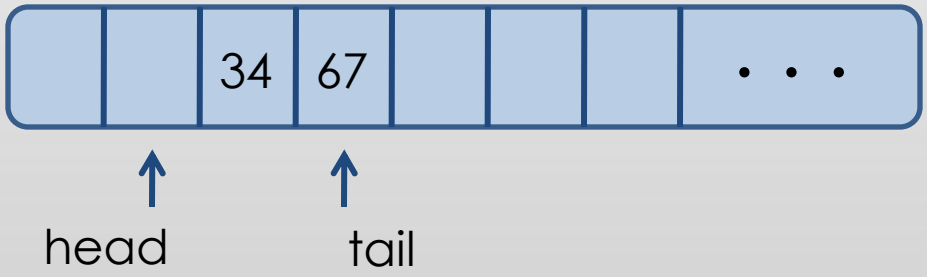
x = 23



Queue – Array Implementation

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

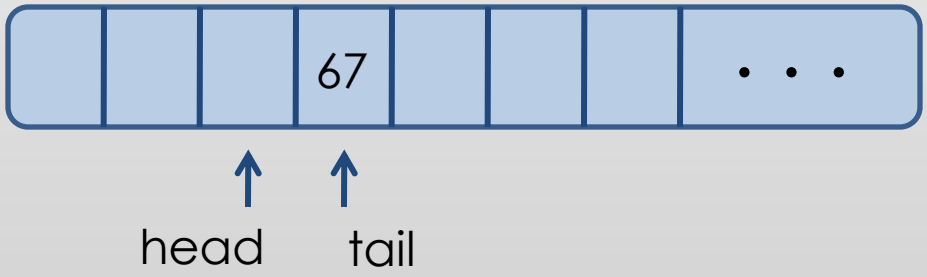
x = 23



Queue – Array Implementation

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

x = 34

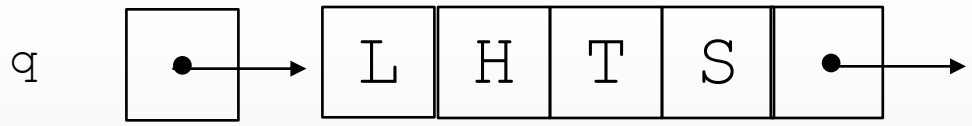


Array Implementation

```
typedef struct node{
    int limit;
    int head;
    int tail;
    int size;
    int *array;
}queue;

queue *create(int max){
  queue *q;
  q = (queue *)malloc(sizeof(queue));
  q->array = (int *)malloc(sizeof(int)*max);
  q->head = 1;
  q->tail = 0;
  q->size = 0;
  q->limit = max;
  return q;
}
```

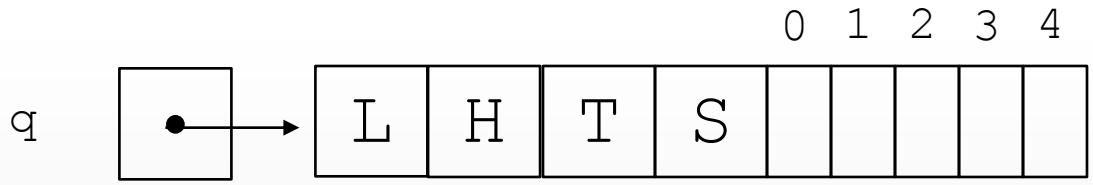
Array Implementation

```
typedef struct node{
    int limit;
    int head;
    int tail;
    int size;
    int *array;
}queue;

queue *create(int max){
  queue *q;
  q = (queue *)malloc(sizeof(queue));
  q->array = (int *)malloc(sizeof(int)*max);
  q->head = 1;
  q->tail = 0;
  q->size = 0;
  q->limit = max;
  return q;
}
```



Array Implementation

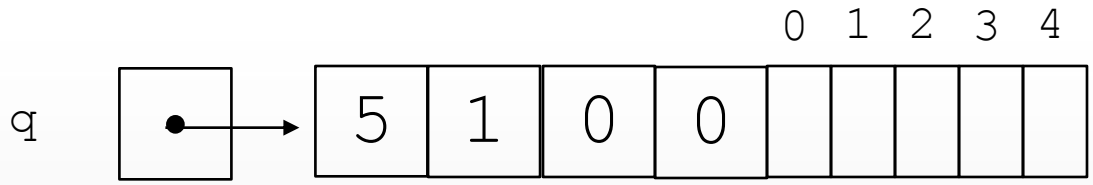
```
typedef struct node{
    int limit;
    int head;
    int tail;
    int size;
    int *array;
}queue;
```




```
queue *create(int max){
  queue *q;
  q = (queue *)malloc(sizeof(queue));
  q->array = (int *)malloc(sizeof(int)*max);
  q->head = 1;
  q->tail = 0;
  q->size = 0;
  q->limit = max;
  return q;
}
```

Array Implementation

```
typedef struct node{
    int limit;
    int head;
    int tail;
    int size;
    int *array;
}queue;
```



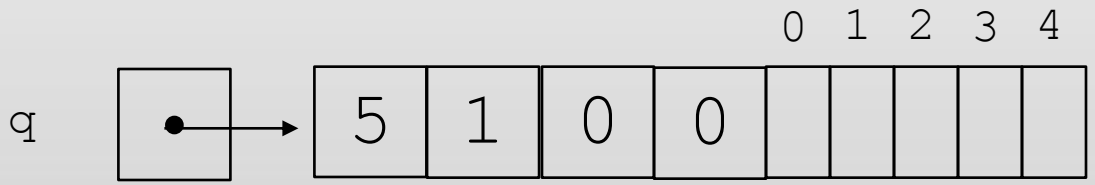

```
queue *create(int max){
  queue *q;
  q = (queue *)malloc(sizeof(queue));
  q->array = (int *)malloc(sizeof(int)*max);
  q->head = 1;
  q->tail = 0;
  q->size = 0;
  q->limit = max;
  return q;
}
```

Array Implementation

```
void enqueue(int x, queue *q){
  if(q->size < q->limit){
    q->size++;
    q->tail = ((q->tail)+1)%LIMIT;
    q->array[q->tail]=x;
  }
  else {
    printf("overflow");
    exit(1);
  }
}
```

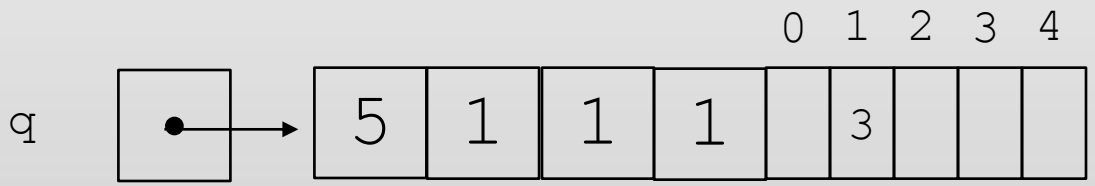
Array Implementation

```
void enqueue(int x, queue *q){
  if(q->size < q->limit){
    q->size++;
    q->tail = ((q->tail)+1)%LIMIT;
    q->array[q->tail]=x;
  }
  else {
    printf("overflow");
    exit(1);
  }
}
```



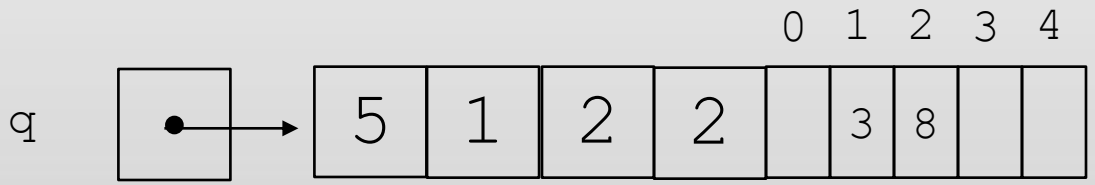
Array Implementation

```
void enqueue(int x, queue *q){
  if(q->size < q->limit){
    q->size++;
    q->tail = ((q->tail)+1)%LIMIT;
    q->array[q->tail]=x;
  }
  else {
    printf("overflow");
    exit(1);
  }
}
```



Array Implementation

```
void enqueue(int x, queue *q){
  if(q->size < q->limit){
    q->size++;
    q->tail = ((q->tail)+1)%LIMIT;
    q->array[q->tail]=x;
  }
  else {
    printf("overflow");
    exit(1);
  }
}
```



Array Implementation

```
int dequeue(queue *q){
  int x;

  if(q->size > 0){
    q->size--;
    x = q->array[q->head]);
    q->head = ((q->head)+1)%LIMIT;
    return x;
  }
  else {
    printf("underflow");
    exit(1);
  }
}
```

Array Implementation

```
int dequeue(queue *q){
  int x;

  if(q->size > 0){
    q->size--;
    x = q->array[q->head]);
    q->head = ((q->head)+1)%LIMIT;
    return x;
  }
  else {
    printf("underflow");
    exit(1);
  }
}
```

Array Implementation

```
int dequeue(queue *q){
  int x;

  if(q->size > 0){
    q->size--;
    x = q->array[q->head]);
    q->head = ((q->head)+1)%LIMIT;
    return x;
  }
  else {
    printf("underflow");
    exit(1);
  }
}
```

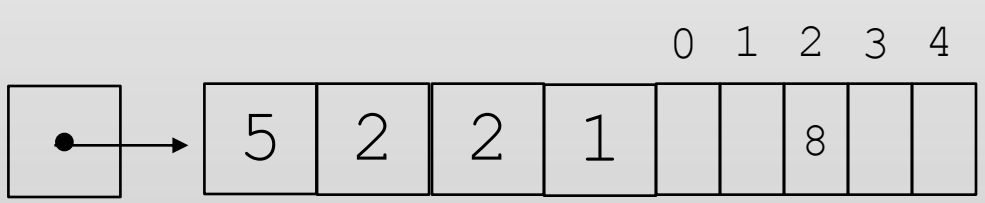
Array Implementation

```
int dequeue(queue *q){
  int x;

  if(q->size > 0){
    q->size--;
    x = q->array[q->head]);
    q->head = ((q->head)+1)%LIMIT;
    return x;
  }
  else {
    printf("underflow");
    exit(1);
  }
}
```

x = 3

Array Implementation

```
int dequeue(queue *q){
  int x;

  if(q->size > 0){
    q->size--;
    x = q->array[q->head]);
    q->head = ((q->head)+1)%LIMIT;
    return x;
  }
  else {
    printf("underflow");
    exit(1);
  }
}
```



3. The Queue ADT

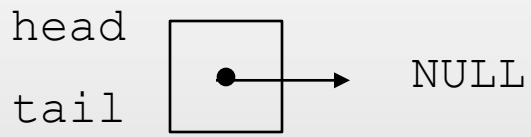
3.2 Linked List Implementation

Linked List Implementation

- linear linked list
- two pointers are needed: head and tail

Linked-list Implementation

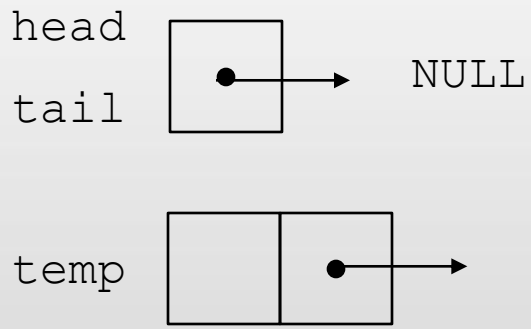
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void enqueue(queue *head, queue *tail,
                int x){
  queue *temp;

  temp=(queue *)malloc(sizeof(queue));
  if(temp==NULL){
    printf("overflow");
    exit(1);
  }
  temp->value=x;
  temp->next=NULL;
  if (tail==NULL)
    head=tail=temp;
  else{
    tail->next=temp;
    tail=temp;
  }
}
```

Linked-list Implementation

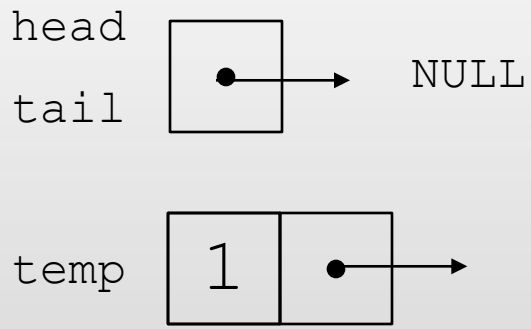
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void enqueue(queue *head, queue *tail,
                int x){
  queue *temp;

  temp=(queue *)malloc(sizeof(queue));
  if(temp==NULL){
    printf("overflow");
    exit(1);
  }
  temp->value=x;
  temp->next=NULL;
  if (tail==NULL)
    head=tail=temp;
  else{
    tail->next=temp;
    tail=temp;
  }
}
```

Linked-list Implementation

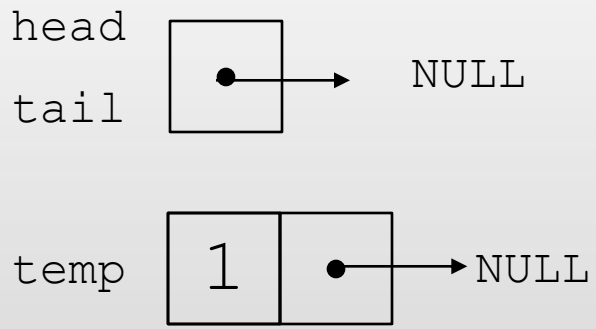
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void enqueue(queue *head, queue *tail,
                int x){
  queue *temp;

  temp=(queue *)malloc(sizeof(queue));
  if(temp==NULL){
    printf("overflow");
    exit(1);
  }
  temp->value=x;
  temp->next=NULL;
  if (tail==NULL)
    head=tail=temp;
  else{
    tail->next=temp;
    tail=temp;
  }
}
```

Linked-list Implementation

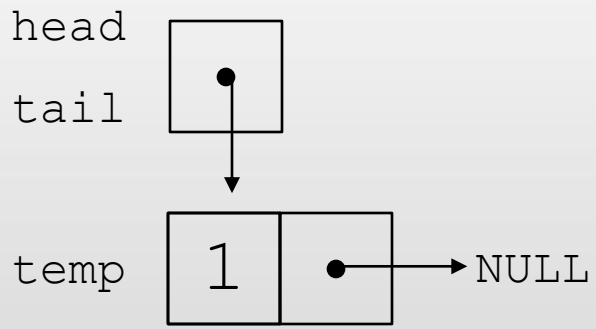
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void enqueue(queue *head, queue *tail,
                int x){
  queue *temp;

  temp=(queue *)malloc(sizeof(queue));
  if(temp==NULL){
    printf("overflow");
    exit(1);
  }
  temp->value=x;
  temp->next=NULL;
  if (tail==NULL)
    head=tail=temp;
  else{
    tail->next=temp;
    tail=temp;
  }
}
```

Linked-list Implementation

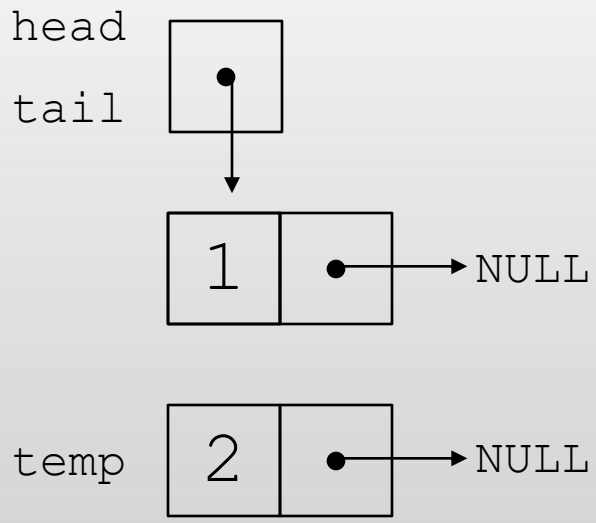
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void enqueue(queue *head, queue *tail,
                int x){
  queue *temp;

  temp=(queue *)malloc(sizeof(queue));
  if(temp==NULL){
    printf("overflow");
    exit(1);
  }
  temp->value=x;
  temp->next=NULL;
  if (tail==NULL)
    head=tail=temp;
  else{
    tail->next=temp;
    tail=temp;
  }
}
```

Linked-list Implementation

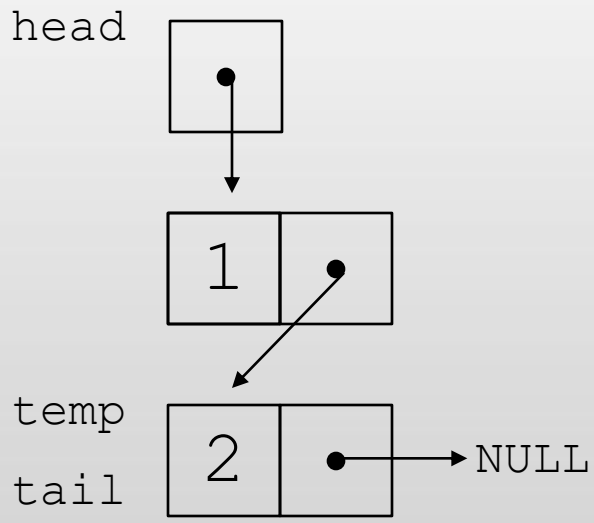
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void enqueue(queue *head, queue *tail,
                int x){
  queue *temp;

  temp=(queue *)malloc(sizeof(queue));
  if(temp==NULL){
    printf("overflow");
    exit(1);
  }
  temp->value=x;
  temp->next=NULL;
  if (tail==NULL)
    head=tail=temp;
  else{
    tail->next=temp;
    tail=temp;
  }
}
```

Linked-list Implementation

```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void enqueue(queue *head, queue *tail,
                int x){
  queue *temp;

  temp=(queue *)malloc(sizeof(queue));
  if(temp==NULL){
    printf("overflow");
    exit(1);
  }
  temp->value=x;
  temp->next=NULL;
  if (tail==NULL)
    head=tail=temp;
  else{
    tail->next=temp;
    tail=temp;
  }
}
```

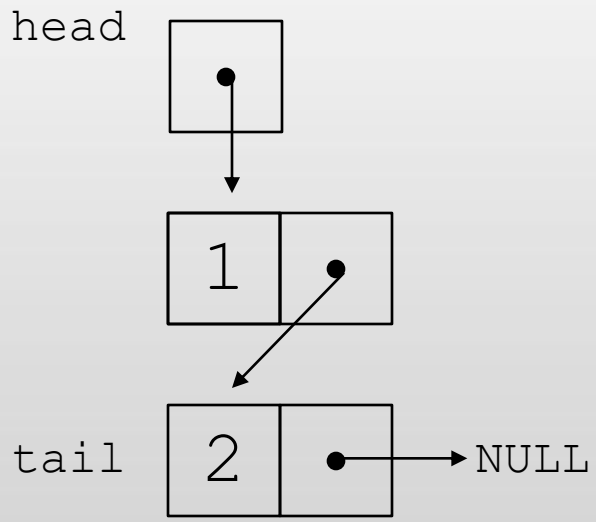
Linked-list Implementation

```
typedef struct node{
  int value;
  struct node *next;
}queue;
```

```
void dequeue(queue *head, queue *tail){
  queue *temp; int x;

  temp=head;
  if(temp==NULL){
    printf("underflow");
    exit(1);
  }
  if(temp->next==NULL)
    head=tail=NULL;
  else{
    head=head->next;
  }
  x=temp->value;
  free(temp);
  return(x);

}
```

Linked-list Implementation

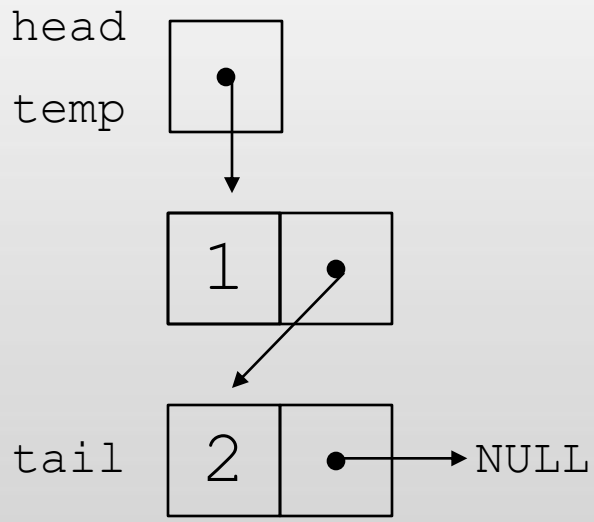
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void dequeue(queue *head, queue *tail){
  queue *temp; int x;

  temp=head;
  if(temp==NULL){
    printf("underflow");
    exit(1);
  }
  if(temp->next==NULL)
    head=tail=NULL;
  else{
    head=head->next;
  }
  x=temp->value;
  free(temp);
  return(x);

}
```

Linked-list Implementation

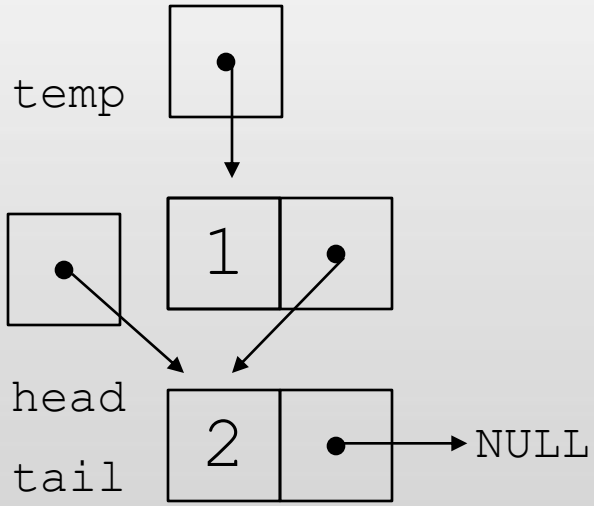
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void dequeue(queue *head, queue *tail){
  queue *temp; int x;

  temp=head;
  if(temp==NULL){
    printf("underflow");
    exit(1);
  }
  if(temp->next==NULL)
    head=tail=NULL;
  else{
    head=head->next;
  }
  x=temp->value;
  free(temp);
  return(x);

}
```

Linked-list Implementation

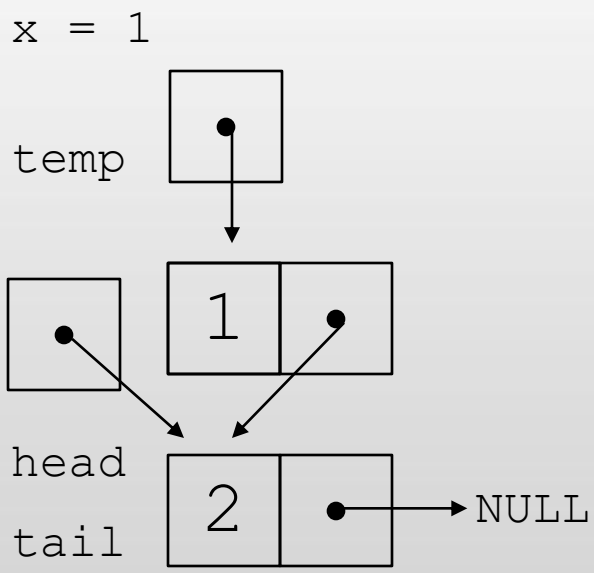
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void dequeue(queue *head, queue *tail){
  queue *temp; int x;

  temp=head;
  if(temp==NULL){
    printf("underflow");
    exit(1);
  }
  if(temp->next==NULL)
    head=tail=NULL;
  else{
    head=head->next;
  }
  x=temp->value;
  free(temp);
  return(x);

}
```

Linked-list Implementation

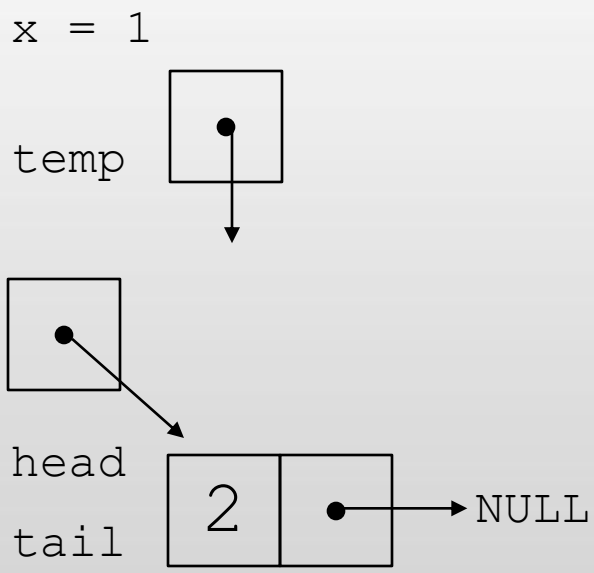
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void dequeue(queue *head, queue *tail){
  queue *temp; int x;

  temp=head;
  if(temp==NULL){
    printf("underflow");
    exit(1);
  }
  if(temp->next==NULL)
    head=tail=NULL;
  else{
    head=head->next;
  }
  x=temp->value;
  free(temp);
  return(x);

}
```

Linked-list Implementation

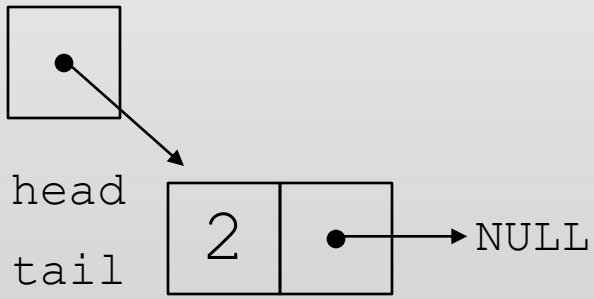
```
typedef struct node{
  int value;
  struct node *next;
}queue;
```




```
void dequeue(queue *head, queue *tail){
  queue *temp; int x;

  temp=head;
  if(temp==NULL){
    printf("underflow");
    exit(1);
  }
  if(temp->next==NULL)
    head=tail=NULL;
  else{
    head=head->next;
  }
  x=temp->value;
  free(temp);
  return(x);

}
```

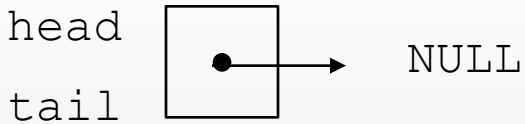
Linked-list Implementation

```
typedef struct node{
  int value;
  struct node *next;
}queue;
```



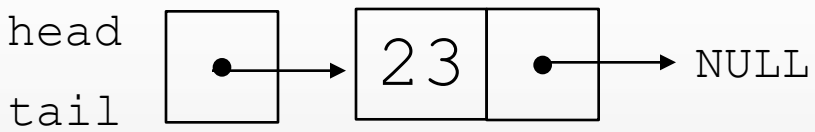

```
void dequeue(queue *head, queue *tail){
  queue *temp; int x;

  temp=head;
  if(temp==NULL){
    printf("underflow");
    exit(1);
  }
  if(temp->next==NULL)
    head=tail=NULL;
  else{
    head=head->next;
  }
  x=temp->value;
  free(temp);
  return(x);

}
```

Linked-list Implementation



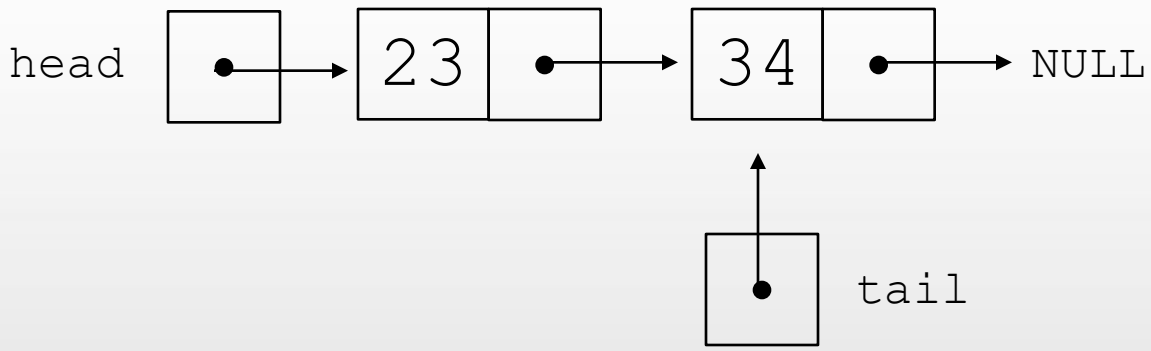

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

Linked-list Implementation



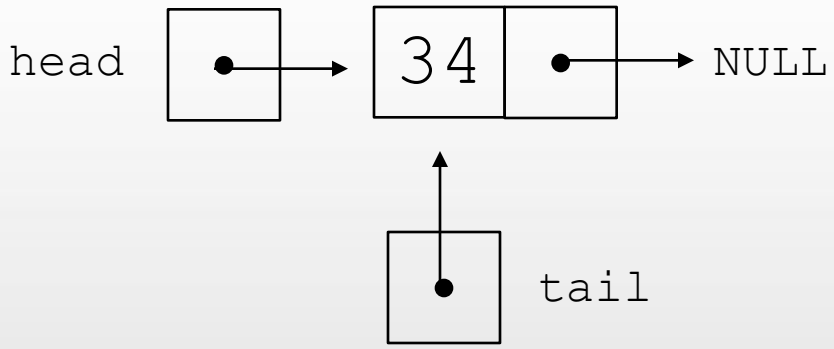

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

Linked-list Implementation




```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

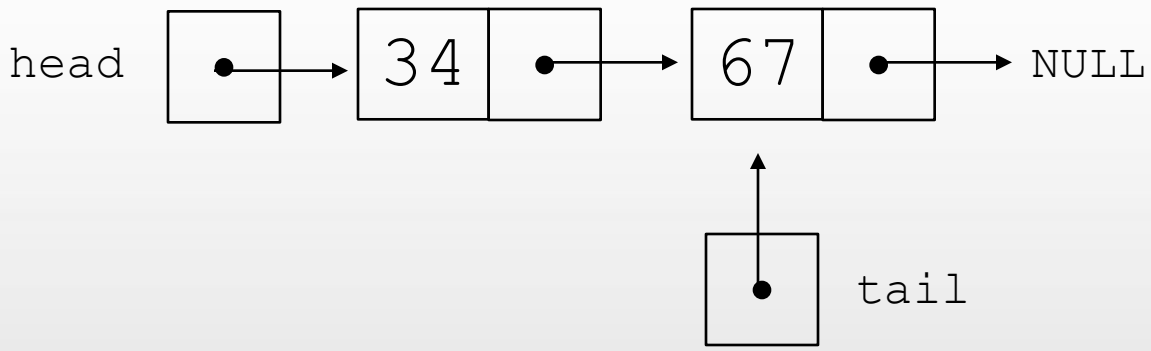
Linked-list Implementation




```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

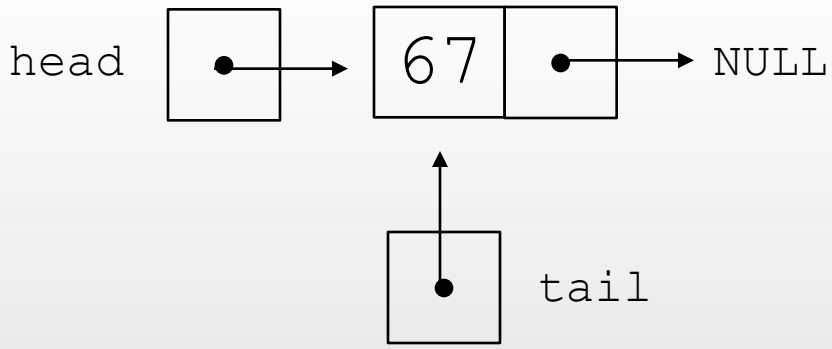
x=23

Linked-list Implementation




```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

Linked-list Implementation



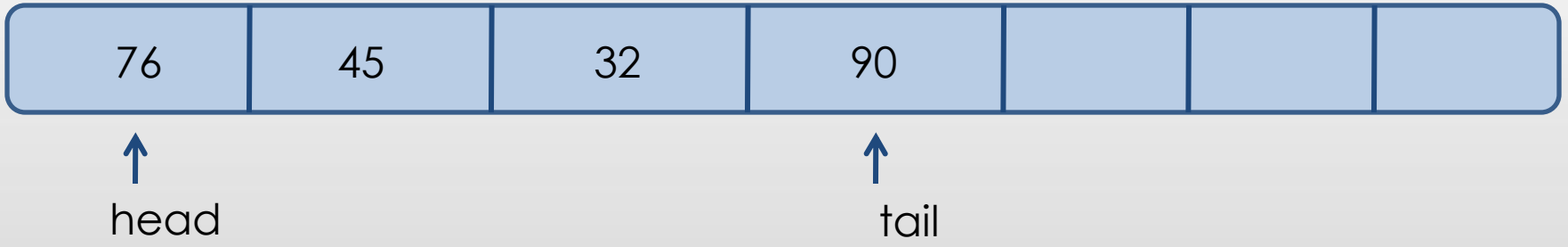

```
enqueue(23);
enqueue(34);
x=dequeue();
enqueue(67);
x=dequeue();
```

x=34

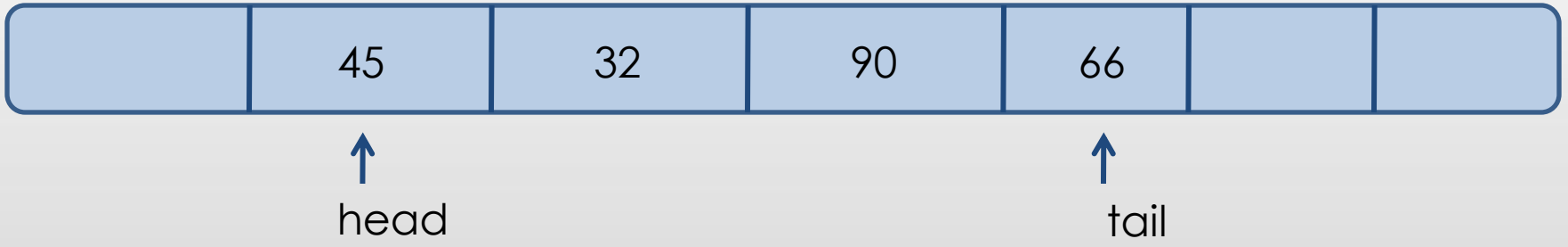
Applications

- Time-Sharing System Simulation
  - Given: *n* user processes in a queue containing the time required to execute the process to completion
  - Simulator algorithm:
    - Process at head is dispatched to use the CPU for a maximum of 10 ticks
    - If a process has remaining execution time of x<10, it is terminated after x ticks, otherwise, the process executes for 10 ticks, its remaining time is reduced by 10 and the process is timed out

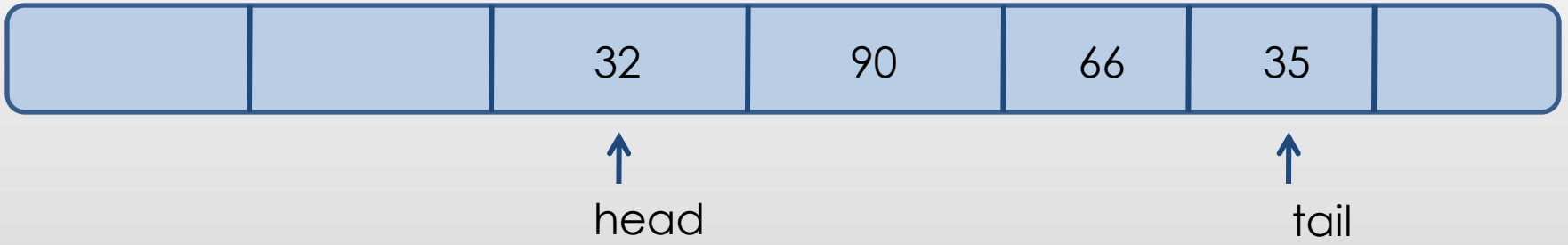
Example



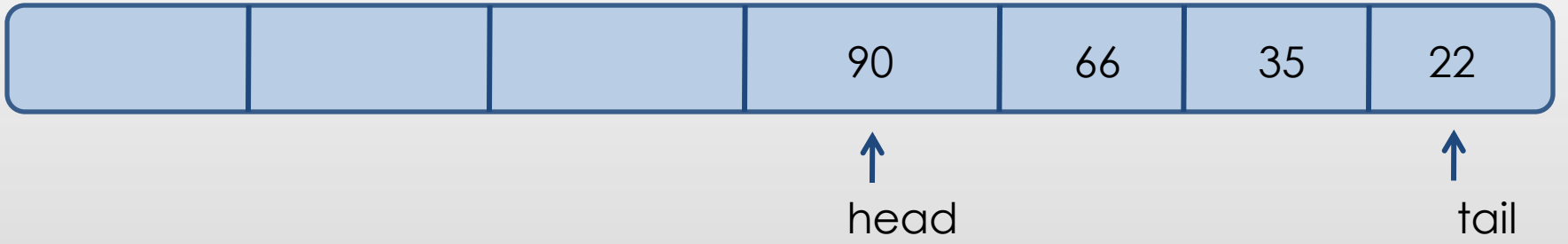
Example



Example



Example



Applications

- Parking Space Simulation
  - Operates in FIFO basis
  - Has a single-lane space
  - Before a vehicle is admitted, its departure time should be later than the last vehicle admitted, otherwise, it will be denied access
  - First car is always admitted
  - Every hour, one car will request to park

Example

Parking Space! OPEN from 8am – 10pm only

9	11	13	14	15	19	20

head

tail

Double-Ended Queues

- Deque – data structure consisting of a list of items, on which the following operations are possible
  - push(x,d)
  - pop(d)
  - inject(x,d)
  - eject(d)

Double-Ended Queues

- push(x,d)
  - Insert item x in front of deque d
- pop(d)
  - Remove front item from deque d and return it
- inject(x,d)
  - Insert item x at the end of deque d
- eject(d)
  - Remove rear item from deque d and return it