NWEN241
Dynamic Data Structures II

Qiang Fu

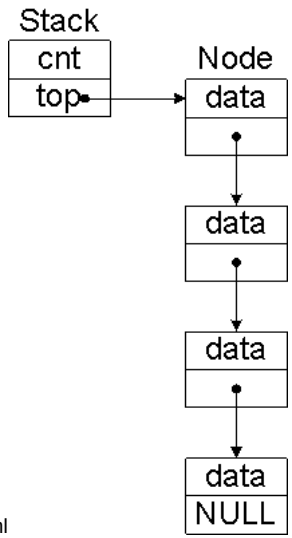
School of Engineering and Computer Science
Victoria University of Wellington

This Lecture

- A bit about stacks
- Queues

Stacks

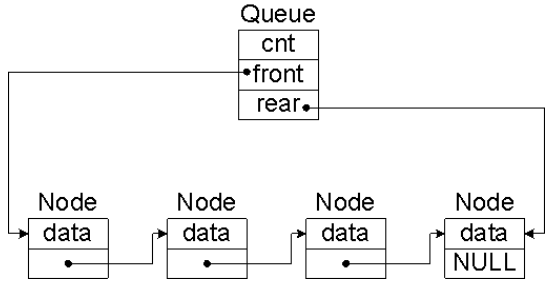
- A stack can be implemented as a linked list
- A stack has access restricted to the top of the list
- Insertion and deletion occur only at the top
- **push** for insertion
- **pop** for deletion
- A stack is a last-in-first-out (LIFO) data structure



http://www.cs.armstrong.edu/liang/animation/web/Stack.html

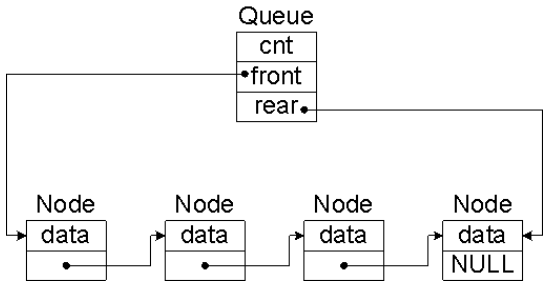
Queues

- A queue can be implemented as a linked list
- A queue has a front and a rear
- Insertion occurs at the rear of the list
- Deletion occurs at the front of the list
- A queue is a *first-in-first-out* (FIFO) data structure



Implementing a Queue

- Structure a queue
  - A header node (Queue)
  - A list of linked nodes
  - The header node has two pointers pointing to the front and the rear of the linked list
  - The header node has a counter counting the number of nodes in the linked list



Implementing a Queue

- The features that a functional queue should have:
  - Insert a node

Implementing a Queue

- The features that a functional queue should have:
  - Insert a node
  - Delete a node

Implementing a Queue

- The features that a functional queue should have:
  - Insert a node
  - Delete a node
  - Test whether the queue is empty

Implementing a Queue

- The features that a functional queue should have:
  - Insert a node
  - Delete a node
  - Test whether the queue is empty
  - Test whether the queue is full

Implementing a Queue

- Functionalities that we should implement:
  - Initialise the queue:
  - test whether the queue is empty:
  - test whether the queue is full:
  - Insert a node (enqueue):
  - Delete a node (dequeue):

Implementing a Queue

- Functionalities that we should implement:
  - Initialise the queue: assign the appropriate values to the header node
  - test whether the queue is empty:
  - test whether the queue is full:
  - Insert a node (enqueue):
  - Delete a node (dequeue):

Implementing a Queue

- Functionalities that we should implement:
  - Initialise the queue: assign the appropriate values to the header node
  - test whether the queue is empty: inspect the header node
  - test whether the queue is full:
  - Insert a node (enqueue):
  - Delete a node (dequeue):

Implementing a Queue

- Functionalities that we should implement:
  - Initialise the queue: assign the appropriate values to the header node
  - test whether the queue is empty: inspect the header node
  - test whether the queue is full: inspect the header node
  - Insert a node (enqueue):
  - Delete a node (dequeue):

Implementing a Queue

- Functionalities that we should implement:
  - Initialise the queue: assign the appropriate values to the header node
  - test whether the queue is empty: inspect the header node
  - test whether the queue is full: inspect the header node
  - Insert a node (enqueue): extend the list and update the header
  - Delete a node (dequeue):

Implementing a Queue

- Functionalities that we should implement:
  - Initialise the queue: assign the appropriate values to the header node
  - test whether the queue is empty: inspect the header node
  - test whether the queue is full: inspect the header node
  - Insert a node (enqueue): extend the list and update the header
  - Delete a node (dequeue): shorten the list and update the header and free the memory allocated to the deleted node

Implementing a Queue

- Add necessary preprocessing directives

```
#include <stdio.h>
#include <stdlib.h> /* malloc ... */

#define Node_Size sizeof(Node)
#define EMPTY 0   /* empty list */
#define FULL 1000 /* maximum 1000 nodes */
```

Implementing a Queue

- Node template (list node)

```
typedef struct node
{ char data;         /* data field */
  struct node *next; /* point to next node */
} Node;
typedef char Data;
typedef Node *ptrNode;
```

- Node template (header node)

```
typedef struct queue
{ int cnt;      /* count the number of nodes */
  ptrNode front;/* point to the front of the list */
  ptrNode rear; /* point to the rear of the list */
} Queue;
typedef Queue *ptrQueue;
```


Implementing a Queue

- Make a boolean type

```
typedef enum boolean {FALSE, TRUE} bool;

/* to be used to ....??? */
```


Implementing a Queue

- Make a boolean type

```
typedef enum boolean {FALSE, TRUE} bool;

/* to be used to check whether the queue */
/* is full, empty */
```


Implementing a Queue

- Functions that we need to have

```
void initialise(ptrQueue);
  /* initialise the queue (header node) */
bool isempty(ptrQueue);
  /* is the queue empty */
bool isfull(ptrQueue);
  /* is the queue full */
void enqueue(Data, ptrQueue);
  /* insert a new node and no return */
Data dequeue(ptrQueue);
  /* delete a node and return the data in */
  /* the node */
```

Implementing a Queue

- Initialisation

```
void initialise(ptrQueue q)
{
  q->cnt = 0;
  q->front = NULL;
  q->rear = NULL;
}
```


Implementing a Queue

- Is the queue full, or is it empty?

```
bool isfull(ptrQueue q) {
  return((bool)(q->cnt==FULL));
}

bool isempty(ptrQueue q) {
  return((bool)(q->cnt==EMPTY));
}
```


Implementing a Queue

- Delete a node

```
Data dequeue(ptrQueue q)
{ /* delete a node in the front and return its data */
  ...
  Data d;         /* to be used to store the data */
  ptrNode tmp;    /* to be used for ... */

  tmp = q->front; /* tmp has the address of the node */
                  /* that is going to be deleted */
  d = tmp->data;
  q->front = tmp->next; /* what is this for? */
  q->cnt--;
  free(tmp);
  return d;
}
```


Implementing a Queue

- Delete a node

```
Data dequeue(ptrQueue q)
{ /* delete a node in the front and return its data */
  ...
  Data d;         /* to be used to store the data */
  ptrNode tmp;    /* to be used for handling the node*/
                  /* that is going to be deleted */
  tmp = q->front; /* tmp has the address of the node */
                  /* that is going to be deleted */
  d = tmp->data;
  q->front = tmp->next; /* what is this for? */
  q->cnt--;
  free(tmp);
  return d;
}
```

Implementing a Queue

- Delete a node

```
Data dequeue(ptrQueue q)
{ /* delete a node in the front and return its data */
  ...
  Data d;         /* to be used to store the data */
  ptrNode tmp;    /* to be used for handling the node*/
                  /* that is going to be deleted */
  tmp = q->front; /* tmp has the address of the node */
                  /* that is going to be deleted */
  d = tmp->data;
  q->front = tmp->next; /*updating the new front node*/
  q->cnt--;
  free(tmp);
  return d;
}
```


Implementing a Queue

- Delete a node

```
Data dequeue(ptrQueue q)
{ /* delete a node in the front and return its data */
  ...
  Data d;         /* to be used to store the data */
  ptrNode tmp;    /* to be used for handling the node*/
                  /* that is going to be deleted */
  tmp = q->front; /* tmp has the address of the node */
                  /* that is going to be deleted */
  d = tmp->data;
  q->front = tmp->next; /*updating the new front node*/
  q->cnt--;
  free(tmp);      /* do not forget this */
  return d;
}
```


Implementing a Queue

- Insert a node (one way)

```
void enqueue(Data d, ptrQueue q)
{ if(isfull(q))              /* if full, abort */
    abort();

  if(isempty(q)) {           /* the very first node */
    q->rear = malloc(Node_Size); /* request memory for a new node */
    q->rear->data = d;
    q->rear->next = NULL;
    q->front = q->rear; /* front and rear point to the same node */
  } else {
    q->rear->next = malloc(Node_Size); /*who is pointing the nu node*/
    q->rear = q->rear->next;  /* rear is now pointing to ... */
    q->rear->data = d;
    q->rear->next = NULL;     /* who is pointing to NULL? */
  }
  q->cnt++;                  /* update the counter in header node */
}
```


Implementing a Queue

- Insert a node (another way - recommended)

```
void enqueue(Data d, ptrQueue q)
{ ...
  ptrNode pn;       /* to create a new node */
  pn = malloc(Node_Size);
  pn->data = d;
  pn->next = NULL;  /* the new node points to NULL */
                    /* who is pointing to the new node? */









}
```

Implementing a Queue

- Insert a node (another way - recommended)

```
void enqueue(Data d, ptrQueue q)
{ ...
  ptrNode pn;       /* to create a new node */
  pn = malloc(Node_Size);
  pn->data = d;
  pn->next = NULL;  /* the new node points to NULL */
                    /* no one yet */
  if(isempty(q)) {  /* the very first node */
    q->front = q->rear = pn;
  } else {
    q->rear->next = pn;    /* who is pointing to who? */
    q->rear = pn;
  }
  q->cnt++;
}
```


Implementing a Queue

- Insert a node (another way - recommended)

```
void enqueue(Data d, ptrQueue q)
{ ...
  ptrNode pn;       /* to create a new node */
  pn = malloc(Node_Size);
  pn->data = d;
  pn->next = NULL;  /* the new node points to NULL */
                    /* no one yet */
  if(isempty(q)) {  /* the very first node */
    q->front = q->rear = pn;
  } else {
    q->rear->next = pn;    /* old rear to new rear node*/
    q->rear = pn;   /* update rear ptr of new rear node */
  }
  q->cnt++;
}
```


Implementing a Queue

- Let us create two queues to sort the uppercase alphabetic characters imported from a file
- Each node holds a character
- Characters with odd values will be stored in the “odd queue”
- Characters with even values will be stored in the “even queue”
- Print the characters and deallocate the memory back to …???

Implementing a Queue

- In main()

```
Data letter;
    /* temporary holder of characters */
Queue evenq, oddq;
    /* create two queues (header nodes) */
ptrQueue even = &evenq, odd = &oddq;
    /* pointers to the header nodes */

initialise(even);
initialise(odd);
```

Implementing a Queue

- In main()

```
while ((letter = getchar())!= EOF) {
    /* read from a file until end-of-file */

  if ((int)letter%2)    /* sorting... */
    enqueue(letter, odd);
  else enqueue(letter, even);
}
```


Implementing a Queue

- In main()

```
while (!isempty(odd)) {
  letter = dequeue(odd);
  printf("%c, %d\n", letter, letter);
}

while (!isempty(even)) {
  letter = dequeue(even);
  printf("%c, %d\n", letter, letter);
}
```


Implementing a Queue

```
% ./a.out < alpha_char
printing character with ...
A, 65
C, 67
E, 69
G, 71
I, 73
K, 75
M, 77
O, 79
Q, 81
S, 83
U, 85
W, 87
Y, 89
```

```
printing characters with
  even integer values...
B, 66
D, 68
F, 70
H, 72
J, 74
L, 76
N, 78
P, 80
R, 82
T, 84
V, 86
X, 88
Z, 90
```


Next Lecture

- Low level programming