Lecture#9

Dynamic Memory Allocation
and
Data Structures II

CENG 102- Algorithms and Programming II,
2024-2025, Spring

Contains materials from:
P. Deitel, H. Deitel, ''C How to Program with an Introduction to C++'', 8th edition, Pearson

12.5 Stacks

- A stack can be implemented as a constrained version of a linked list.
- New nodes can be added to a stack and removed from a stack *only* at the *top*.
- For this reason, a stack is referred to as a last-in, first-out (LIFO) data structure.
- A stack is referenced via a pointer to the top element of the stack.
- The link member in the last node of the stack is set to `NULL` to indicate the bottom of the stack.

12.5 Stacks (Cont.)

- Figure 12.7 illustrates a stack with several nodes
  - `stackPtr` points to the stack's top element.
- Stacks and linked lists are represented identically.
- The difference between stacks and linked lists is that insertions and deletions may occur *anywhere* in a linked list, but *only* at the *top* of a stack.

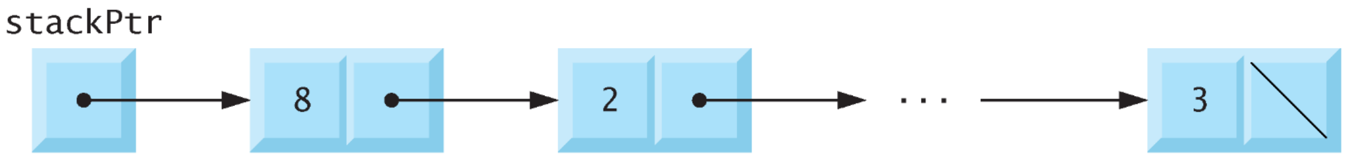


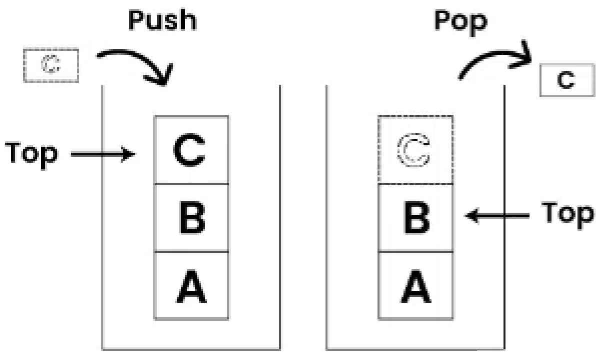
Fig. 12.7 | Stack graphical representation.

Common Programming Error 12.5

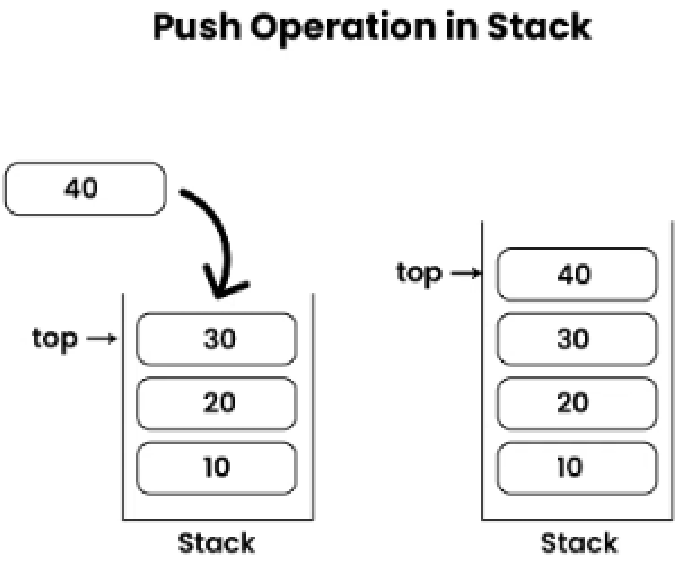
Not setting the link in the bottom node of a stack to `NULL` *can lead to runtime errors.*

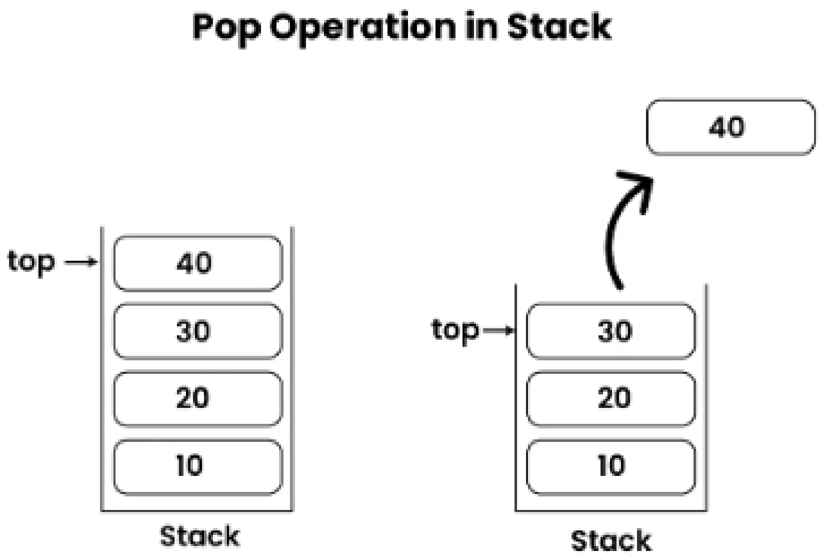
12.5 Stacks (Cont.)

- The primary functions used to manipulate a stack are `push` and `pop`.
- Function `push` creates a new node and places it on top of the stack.
- Function `pop` *removes* a node from the *top* of the stack, *returns the popped value* and *frees* the memory that was allocated to the popped node.



12.5 Stacks (Cont.)





12.5 Stacks (Cont.)

- Figure 12.8 (output shown in Fig. 12.9) implements a simple stack of integers.
- The program provides three options:
  - 1) *push* a value onto the stack (function `push`)
  - 2) *pop* a value off the stack (function `pop`)
  - 3) terminate the program.

```c
// Fig. 12.8: fig12_08.c
// A simple stack program
#include <stdio.h>
#include <stdlib.h>

// self-referential structure
struct stackNode {
   int data; // define data as an int
   struct stackNode *nextPtr; // stackNode pointer
};

typedef struct stackNode StackNode; // synonym for struct stackNode
typedef StackNode *StackNodePtr; // synonym for StackNode*

// prototypes
void push(StackNodePtr *topPtr, int info);
int pop(StackNodePtr *topPtr);
int isEmpty(StackNodePtr topPtr);
void printStack(StackNodePtr currentPtr);
void instructions(void);

```

Fig. 12.8 | A simple stack program. (Part 1 of 7.)

```c
// function main begins program execution
int main(void)
{
   StackNodePtr stackPtr = NULL; // points to stack top
   int value; // int input by user

   instructions(); // display the menu
   printf("%s", "? ");
   unsigned int choice; // user's menu choice
   scanf("%u", &choice);

   // while user does not enter 3
   while (choice != 3) {

      switch (choice) {
         // push value onto stack
         case 1:
            printf("%s", "Enter an integer: ");
            scanf("%d", &value);
            push(&stackPtr, value);
            printStack(stackPtr);
            break;
```

Fig. 12.8 | A simple stack program. (Part 2 of 7.)

```
            // pop value off stack
            case 2:
               // if stack is not empty
               if (!isEmpty(stackPtr)) {
                  printf("The popped value is %d.\n", pop(&stackPtr));
               }

               printStack(stackPtr);
               break;
            default:
               puts("Invalid choice.\n");
               instructions();
               break;
         }

         printf("%s", "? ");
         scanf("%u", &choice);
      }

      puts("End of run.");
   }

```

Fig. 12.8 | A simple stack program. (Part 3 of 7.)

```c
// display program instructions to user
void instructions(void)
{
   puts("Enter choice:\n"
      "1 to push a value on the stack\n"
      "2 to pop a value off the stack\n"
      "3 to end program");
}

// insert a node at the stack top
void push(StackNodePtr *topPtr, int info)
{
   StackNodePtr newPtr = malloc(sizeof(StackNode));

   // insert the node at stack top
   if (newPtr != NULL) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   }
   else { // no space available
      printf("%d not inserted. No memory available.\n", info);
   }
}
```

Fig. 12.8 | A simple stack program. (Part 4 of 7.)

```c

// remove a node from the stack top
int pop(StackNodePtr *topPtr)
{
    StackNodePtr tempPtr = *topPtr;
    int popValue = (*topPtr)->data;
    *topPtr = (*topPtr)->nextPtr;
    free(tempPtr);
    return popValue;
}

```

Fig. 12.8 | A simple stack program. (Part 5 of 7.)

```
// print the stack
void printStack(StackNodePtr currentPtr)
{
   // if stack is empty
   if (currentPtr == NULL) {
      puts("The stack is empty.\n");
   }
   else {
      puts("The stack is:");

      // while not the end of the stack
      while (currentPtr != NULL) {
         printf("%d --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      puts("NULL\n");
   }
}

```

Fig. 12.8 | A simple stack program. (Part 6 of 7.)

```c
// return 1 if the stack is empty, 0 otherwise
int isEmpty(StackNodePtr topPtr)
{
    return topPtr == NULL;
}
```

Fig. 12.8 | A simple stack program. (Part 7 of 7.)

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
```

Fig. 12.9 | Sample output from the program of Fig. 12.8. (Part 1 of 2.)

```
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

Fig. 12.9 | Sample output from the program of Fig. 12.8. (Part 2 of 2.)

12.5.1 Function push

- Function push places a new node at the top of the stack.
- The function consists of three steps:
  - Create a new node by calling malloc and assign the location of the allocated memory to newPtr.
  - Assign to newPtr->data the value to be placed on the stack and assign *topPtr (the stack top pointer) to newPtr->nextPtr—the link member of newPtr now points to the previous top node.
  - Assign newPtr to *topPtr—*topPtr now points to the new stack top.

12.5.1 Function push

- Manipulations involving `*topPtr` change the value of `stackPtr` in `main`.
- Figure 12.10 illustrates function `push`.
- Part (a) of the figure shows the stack and the new node before the `push` operation.
- The dotted arrows in part (b) illustrate Steps 2 and 3 of the `push` operation that enable the node containing `12` to become the new stack top.

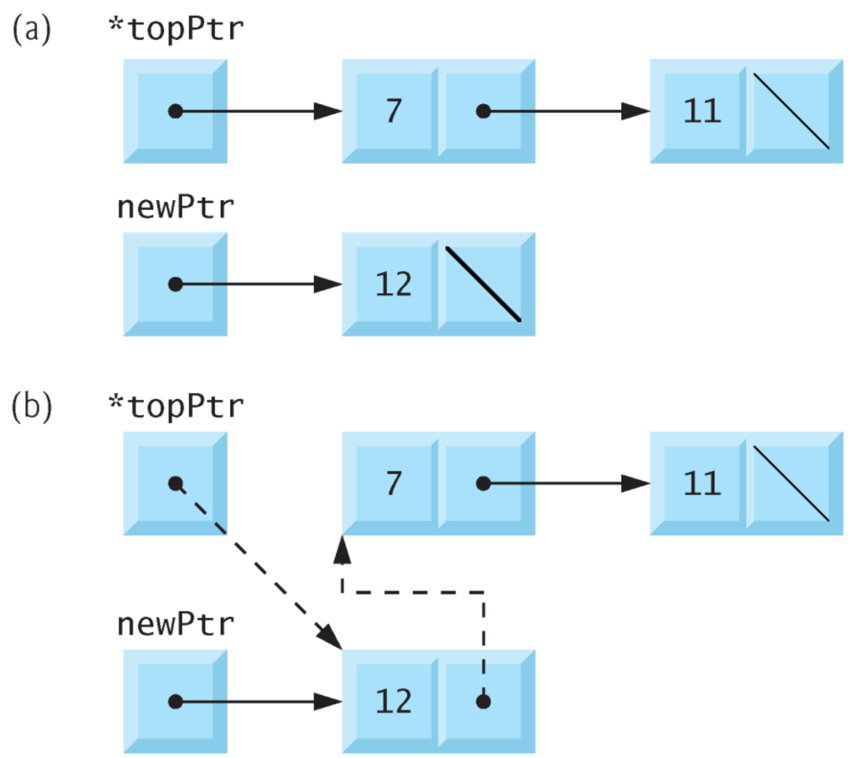


Fig. 12.10 | `push` operation.

12.5.2 Function pop

- Function `pop` removes a node from the top of the stack.
- Function `main` determines if the stack is empty before calling `pop`.
- The `pop` operation consists of five steps:
  - Assign `*topPtr` to `tempPtr`, which will be used to free the unneeded memory
  - Assign `(*topPtr)->data` to `popValue` to *save* the value in the top node
  - Assign `(*topPtr)->nextPtr` to `*topPtr` so `*topPtr` contains *address of the new top node*
  - *Free the memory* pointed to by `tempPtr`
  - *Return* *`popValue`* to the caller

12.5.2 Function pop (Cont.)

- Figure 12.11 illustrates function `pop`.
- Part (a) shows the stack *after* the previous `push` operation.
- Part (b) shows `tempPtr` pointing to the *first node* of the stack and `topPtr` pointing to the *second node* of the stack.
- Function `free` is used to *free the memory* pointed to by `tempPtr`.

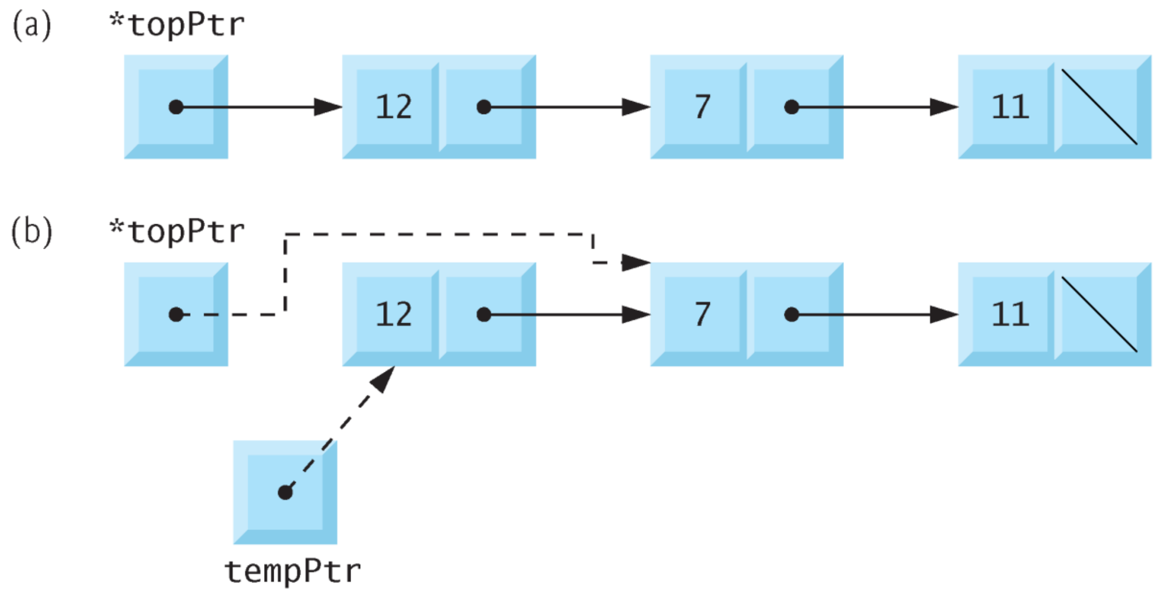


Fig. 12.11 | pop operation.

12.6 Queues

- Another common data structure is the queue.
- A queue is similar to a checkout line in a grocery store—the *first* person in line is *serviced first*, and other customers enter the line only at the *end* and *wait* to be serviced.
- Queue nodes are removed *only* from the head of the queue and are inserted *only* at the tail of the queue.
- For this reason, a queue is referred to as a first-in, first-out (FIFO) data structure.
- The *insert* and *remove* operations are known as `enqueue` and `dequeue`, respectively.

12.6 Queues (Cont.)

- Queues have many applications in computer systems.
- For example, *print spooling*.
- A multiuser environment may have only a single printer.
- Many users may be generating outputs to be printed.
- If the printer is busy, other outputs may still be generated.
- These are spooled to disk where they wait in a *queue* until the printer becomes available.

12.6 Queues (Cont.)

- Another example, information packets also wait in queues in computer networks.
- Each time a packet arrives at a network node, it must be routed to the next node on the network along the path to its final destination.
- The routing node routes one packet at a time, so additional packets are enqueued until the router can route them.
- Figure 12.12 illustrates a queue with several nodes.
- Note the pointers to the head of the queue and the tail of the queue.

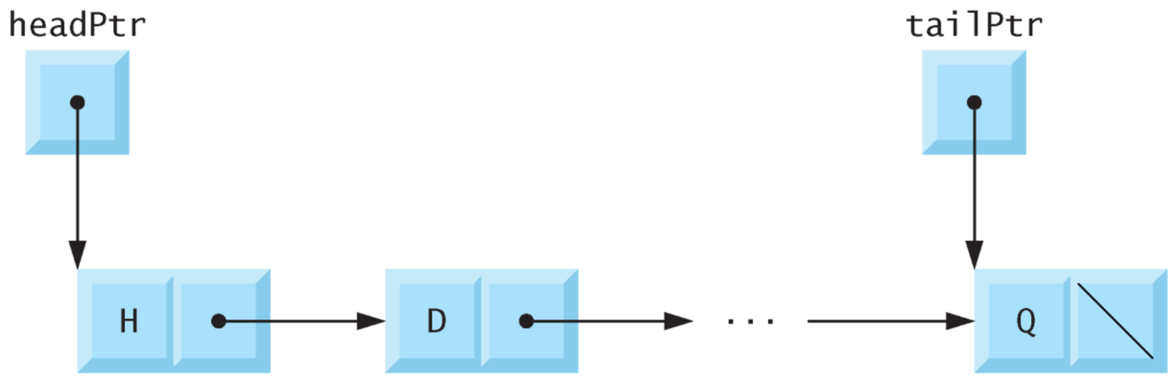
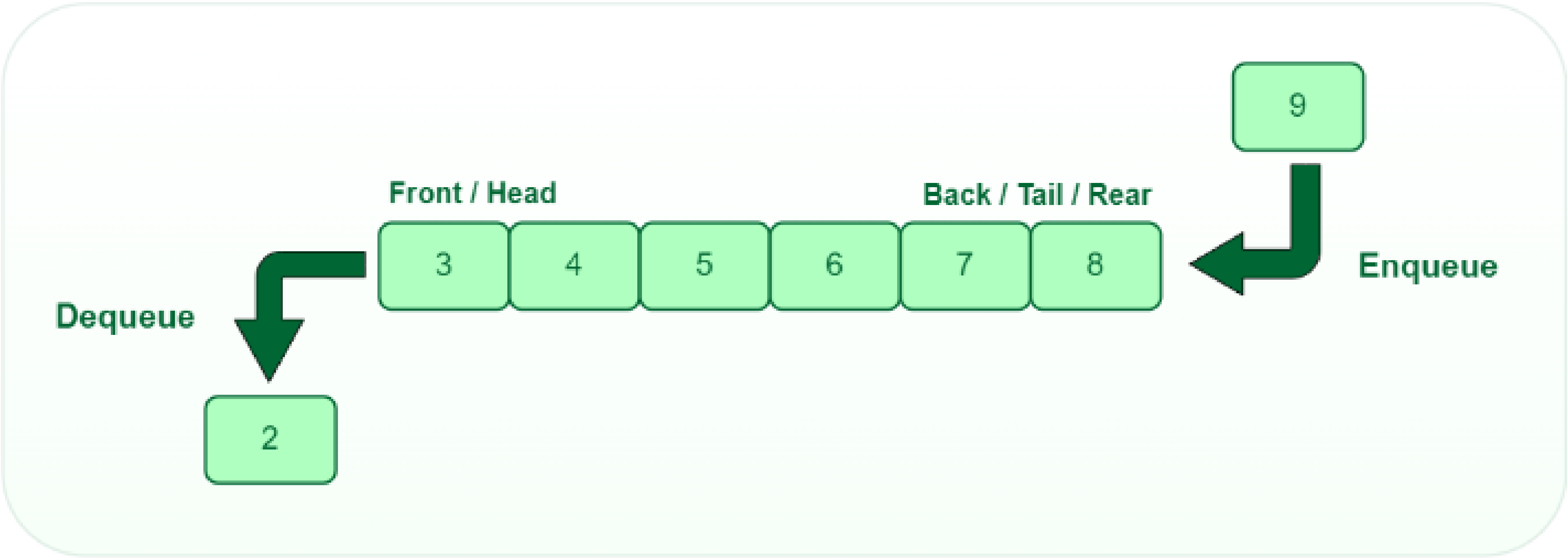


Fig. 12.12 | Queue graphical representation.

Common Programming Error 12.6

Not setting the link in the last node of a queue to `NULL` *can lead to runtime errors.*


9
Front / Head
Back / Tail / Rear
3
4
5
6
7
8
Enqueue
Dequeue
2

12.6 Queues (Cont.)

- Figure 12.13 (output in Fig. 12.14) performs queue manipulations.
- The program provides several options:
  - 1) *insert* a node in the queue (function `enqueue`)
  - 2) *remove* a node from the queue (function `dequeue`)
  - 3) terminate the program.

```c
// Fig. 12.13: fig12_13.c
// Operating and maintaining a queue
#include <stdio.h>
#include <stdlib.h>

// self-referential structure
struct queueNode {
   char data; // define data as a char
   struct queueNode *nextPtr; // queueNode pointer
};

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

// function prototypes
void printQueue(QueueNodePtr currentPtr);
int isEmpty(QueueNodePtr headPtr);
char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr);
void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value);
void instructions(void);

// function main begins program execution
int main(void)
{
```

Fig. 12.13 | Operating and maintaining a queue. (Part 1 of 7.)

```
    QueueNodePtr headPtr = NULL; // initialize headPtr
    QueueNodePtr tailPtr = NULL; // initialize tailPtr
    char item; // char input by user

    instructions(); // display the menu
    printf("%s", "? ");
    unsigned int choice; // user's menu choice
    scanf("%u", &choice);

    // while user does not enter 3
    while (choice != 3) {

       switch(choice) {
          // enqueue value
          case 1:
             printf("%s", "Enter a character: ");
             scanf("\n%c", &item);
             enqueue(&headPtr, &tailPtr, item);
             printQueue(headPtr);
             break;
```

Fig. 12.13 | Operating and maintaining a queue. (Part 2 of 7.)

```
            // dequeue value
            case 2:
               // if queue is not empty
               if (!isEmpty(headPtr)) {
                  item = dequeue(&headPtr, &tailPtr);
                  printf("%c has been dequeued.\n", item);
               }

               printQueue(headPtr);
               break;
            default:
               puts("Invalid choice.\n");
               instructions();
               break;
         }

         printf("%s", "? ");
         scanf("%u", &choice);
      }

      puts("End of run.");
   }

```

Fig. 12.13 | Operating and maintaining a queue. (Part 3 of 7.)

```c
// display program instructions to user
void instructions(void)
{
   printf ("Enter your choice:\n"
           "   1 to add an item to the queue\n"
           "   2 to remove an item from the queue\n"
           "   3 to end\n");
}

```

Fig. 12.13 | Operating and maintaining a queue. (Part 4 of 7.)

```
// insert a node at queue tail
void enqueue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr, char value)
{
   QueueNodePtr newPtr = malloc(sizeof(QueueNode));

   if (newPtr != NULL) { // is space available?
      newPtr->data = value;
      newPtr->nextPtr = NULL;

      // if empty, insert node at head
      if (isEmpty(*headPtr)) {
         *headPtr = newPtr;
      }
      else {
         (*tailPtr)->nextPtr = newPtr;
      }

      *tailPtr = newPtr;
   }
   else {
      printf("%c not inserted. No memory available.\n", value);
   }
}

```

Fig. 12.13 | Operating and maintaining a queue. (Part 5 of 7.)

```c
// remove node from queue head
char dequeue(QueueNodePtr *headPtr, QueueNodePtr *tailPtr)
{
   char value = (*headPtr)->data;
   QueueNodePtr tempPtr = *headPtr;
   *headPtr = (*headPtr)->nextPtr;

   // if queue is empty
   if (*headPtr == NULL) {
      *tailPtr = NULL;
   }

   free(tempPtr);
   return value;
}

// return 1 if the queue is empty, 0 otherwise
int isEmpty(QueueNodePtr headPtr)
{
   return headPtr == NULL;
}

```

Fig. 12.13 | Operating and maintaining a queue. (Part 6 of 7.)

```
// print the queue
void printQueue(QueueNodePtr currentPtr)
{
   // if queue is empty
   if (currentPtr == NULL) {
      puts("Queue is empty.\n");
   }
   else {
      puts("The queue is:");

      // while not end of queue
      while (currentPtr != NULL) {
         printf("%c --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      puts("NULL\n");
   }
}
```

Fig. 12.13 | Operating and maintaining a queue. (Part 7 of 7.)

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end

? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL
```

Fig. 12.14 | Sample output from the program in Fig. 12.13. (Part 1 of 2.)

```
? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.

? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

Fig. 12.14 | Sample output from the program in Fig. 12.13. (Part 2 of 2.)

12.6.1 Function enqueue

- Function `enqueue` receives three arguments from `main`: the address of the *pointer* to the *head of the queue*, the *address* of the p*ointer to the tail of the queue* and the *value* to be inserted in the queue.

12.6 Queues (Cont.)

- The function consists of three steps:
  - To create a new node: Call `malloc`, assign the allocated memory location to `newPtr`, assign the value to be inserted in the queue to `newPtr->data` and assign `NULL` to `newPtr->nextPtr`
  - If the queue is empty, assign `newPtr` to `*headPtr`, because the new node will be both the head and tail of the queue; otherwise, assign pointer `newPtr` to `(*tailPtr)->nextPtr`, because the new node will be placed after the previous tail node.
  - Assign `newPtr` to `*tailPtr`, because the new node is the queue's tail.

12.6 Queues (Cont.)

- Figure 12.15 illustrates an enqueue operation.
- Part (a) shows the queue and the new node *before* the operation.
- The dotted arrows in part (b) illustrate *Steps 2* and *3* of function enqueue that enable a new node to be added to the *end* of a queue that is not empty.

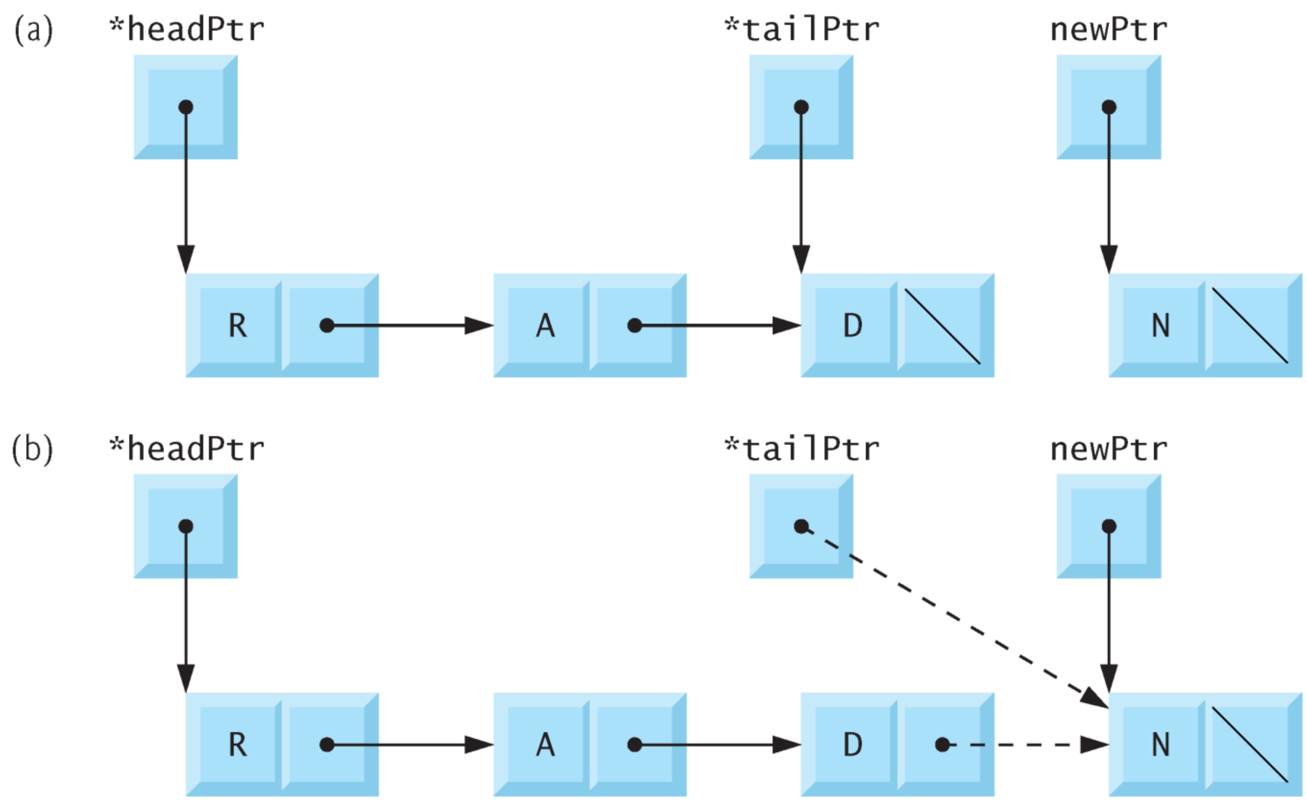


Fig. 12.15 | enqueue operation.

12.6.2 Function `dequeue`

- Function `dequeue` receives the *address* of the *pointer to the head of the queue* and the *address* of the *pointer to the tail of the queue* as arguments and removes the *first* node from the queue.

12.6.2 Function dequeue

- The `dequeue` operation consists of six steps:
  - Assign `(*headPtr)->data` to `value` to save the data
  - Assign `*headPtr` to `tempPtr`, which will be used to `free` the unneeded memory
  - Assign `(*headPtr)->nextPtr` to `*headPtr` so that `*headPtr` now points to the new first node in the queue
  - If `*headPtr` is `NULL`, assign `NULL` to `*tailPtr` because the queue is now empty.
  - Free the memory pointed to by `tempPtr`
  - Return `value` to the caller

12.6 Queues (Cont.)

- Figure 12.16 illustrates function `dequeue`.
- Part (a) shows the queue *after* the preceding `enqueue` operation.
- Part (b) shows `tempPtr` pointing to the *dequeued node*, and `headPtr` pointing to the new *first node* of the queue.
- Function `free` is used to *reclaim the memory* pointed to by `tempPtr`.

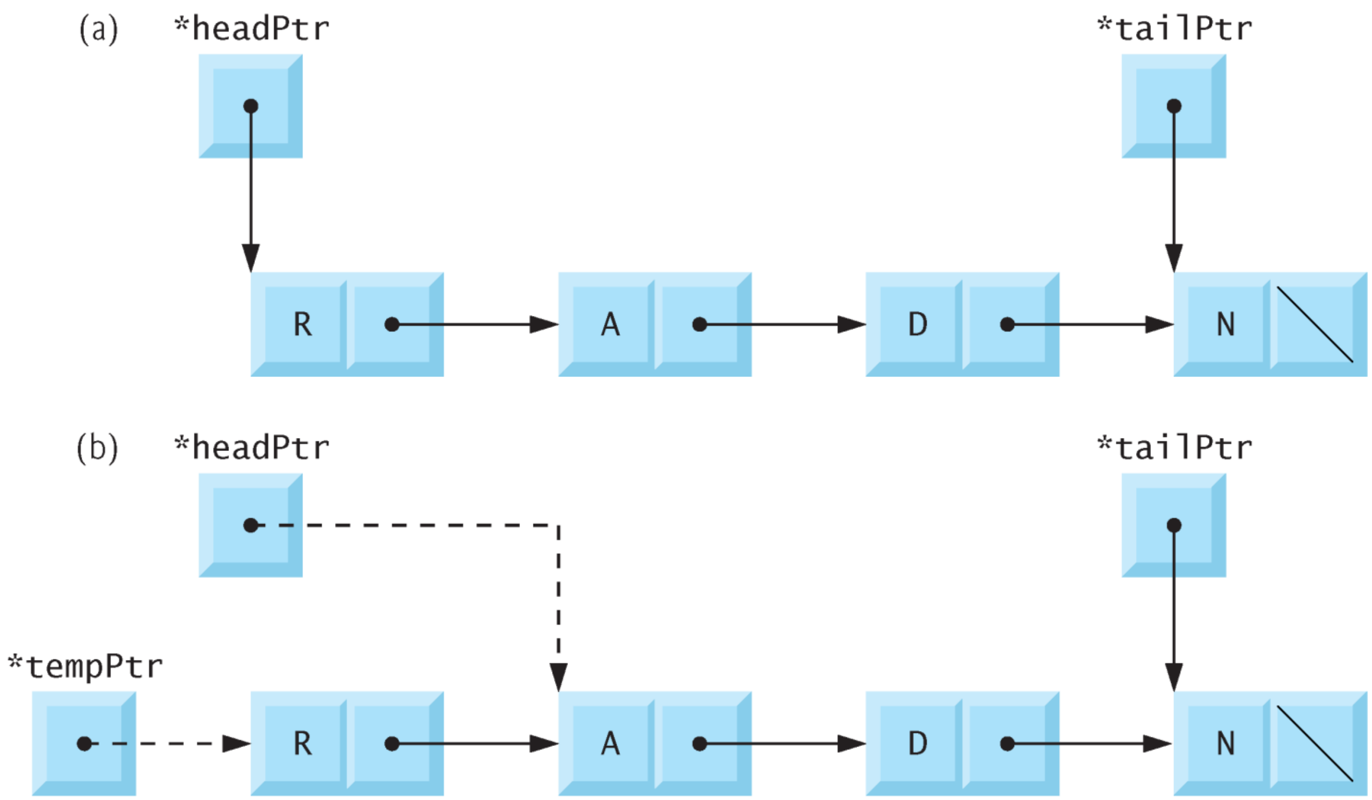


Fig. 12.16 | dequeue operation.