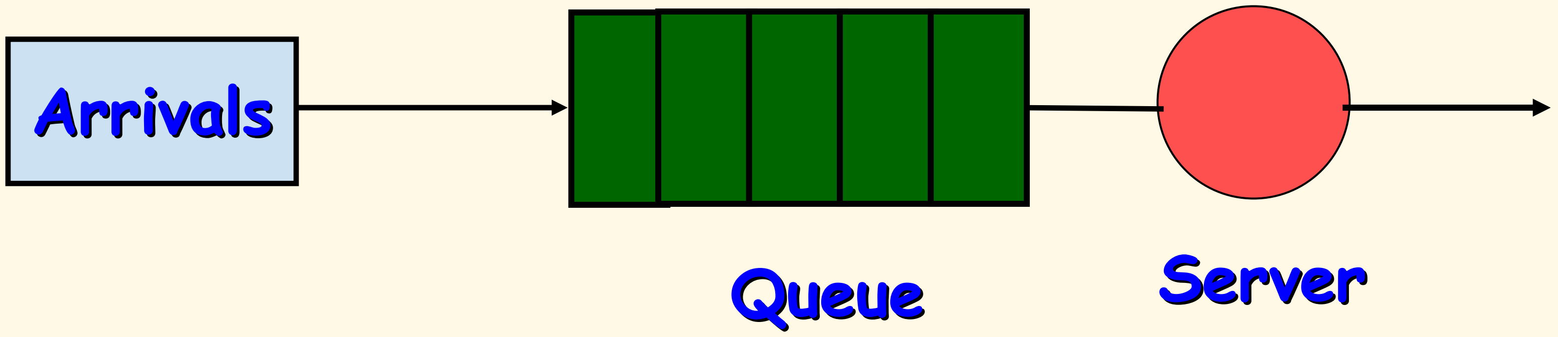
Data Structures

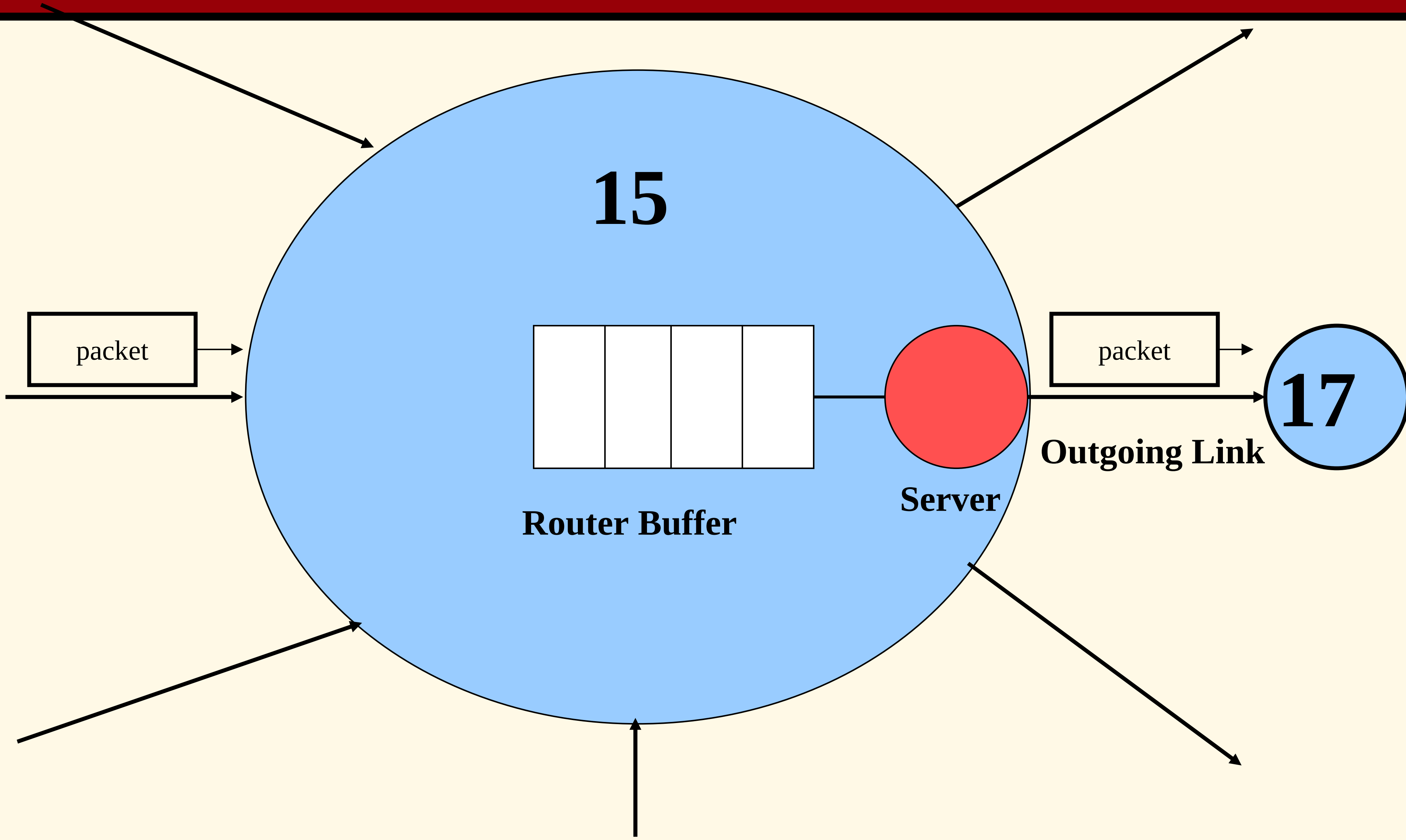
Data Structures

- Queues
  - Queuing System Models
  - Queue Data Structures
  - A Queue Example
- Trees
  - Binary Trees
  - Tree Traversals
    - Inorder, preorder, postorder
- Stacks
    - A Stack Example
- Hashing
- Static Hashing
- Linear probing
- Chaining

Simple Queuing Model



Router Node



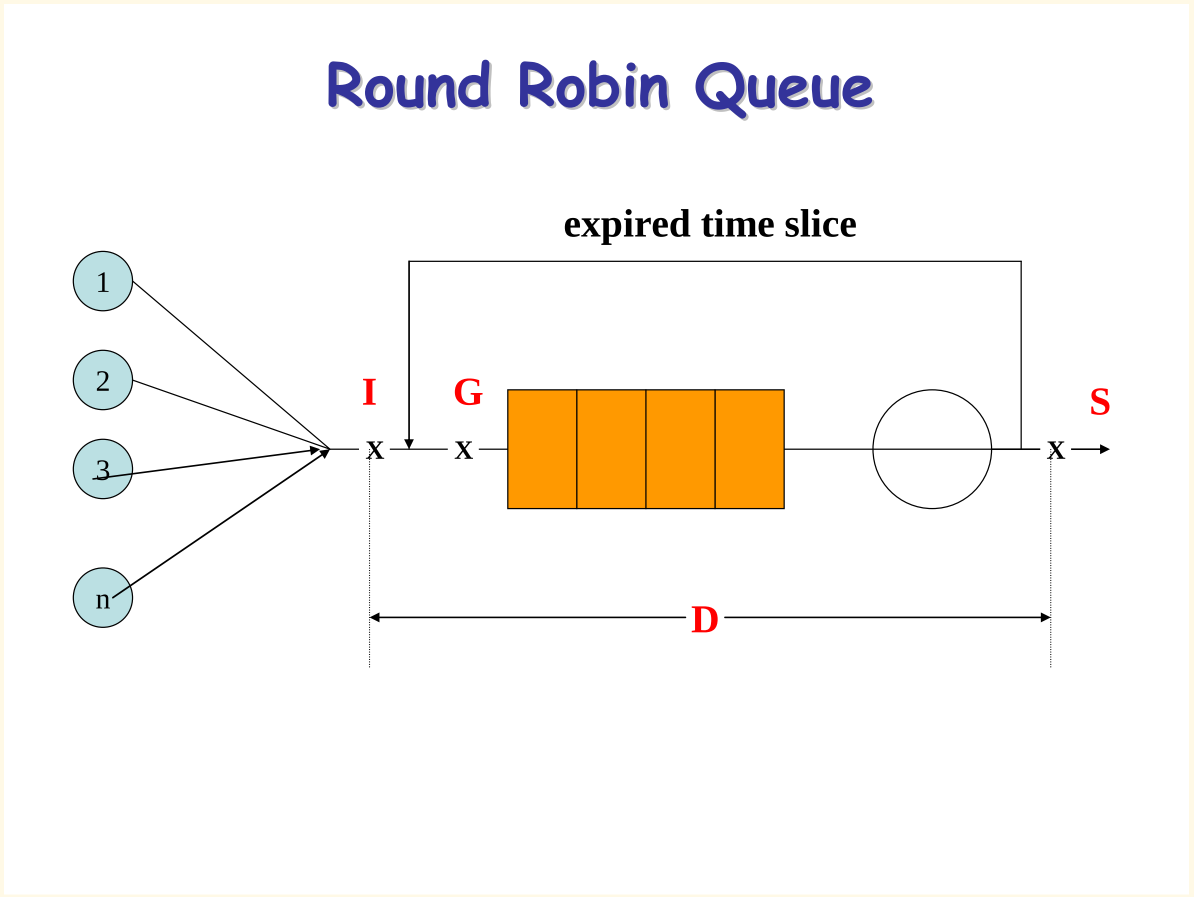
Performance Metrics (General Definitions)

- Utilization :: the percentage of time a device is busy servicing a “customer”.
- Throughput :: the number of jobs processed by the “system” per unit time.
- Response time :: the time required to receive a response to a request (round-trip time).
- Delay :: the time to traverse from one end to the other in a system.

12.6 Queues

- Queues
  - Are very common structures in operating systems and computer networks.
  - The abstraction in system queues is of customers queued with one customer in service.
  - Customers are placed in the queue First-In, first-Out (FIFO).
  - Customers are removed from the head of the queue structure. When modeling a server, the customer at the head of the queue is in service.
  - Customers are inserted at the back (or tail) of the queue.
  - Queues can model either finite of infinite buffer space.

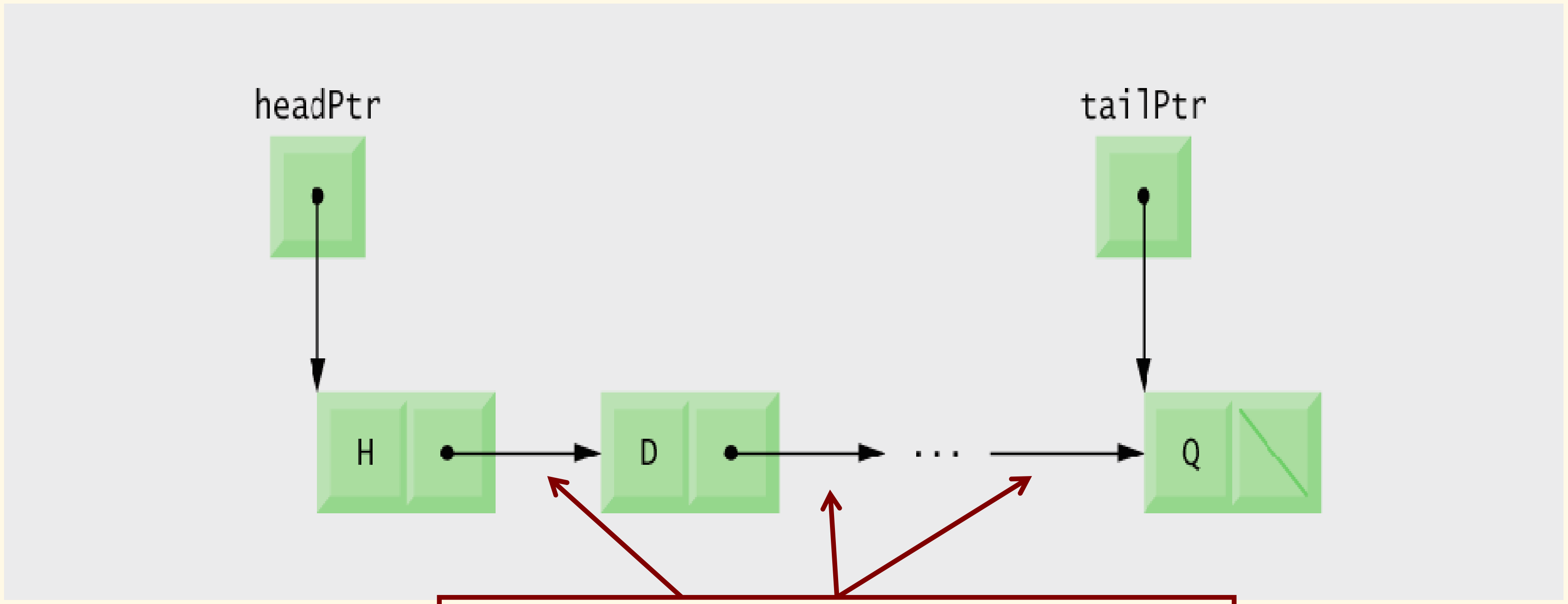
Round Robin Queuing



12.6 Queues

- Queues
  - When modeling a finite buffer, when the buffer is full, an arriving customer is dropped (normally from the tail). This is known as a **drop-tail** queue.
- Queue data structure operations:
  - Insert or **enqueue**
  - Remove or **dequeue**

Fig. 12.12 Queue Graphical Representation



Pointers go from head towards the tail

Fig. 12.15 Enqueue Operation

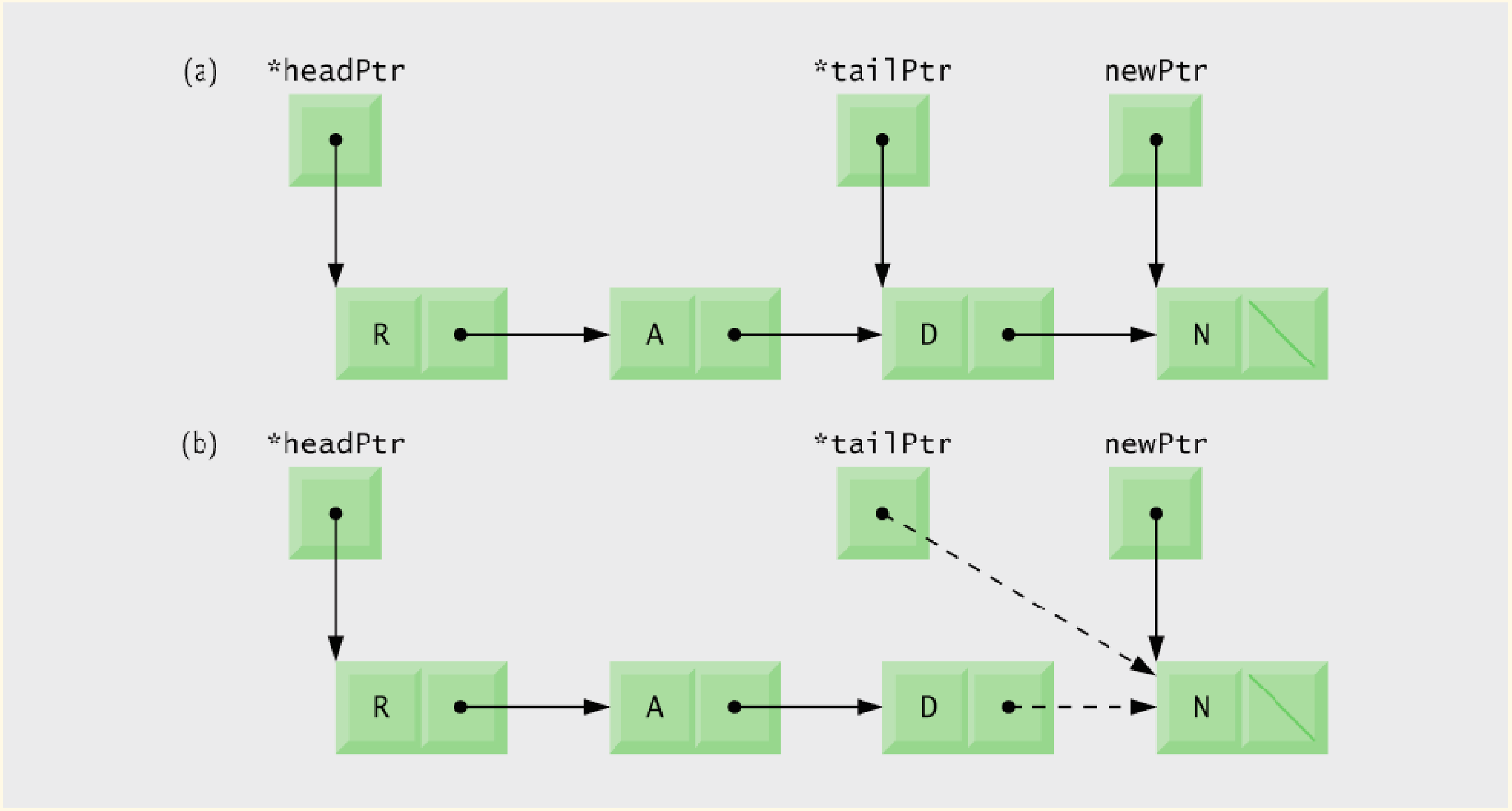


Fig. 12.15 Dequeue Operation

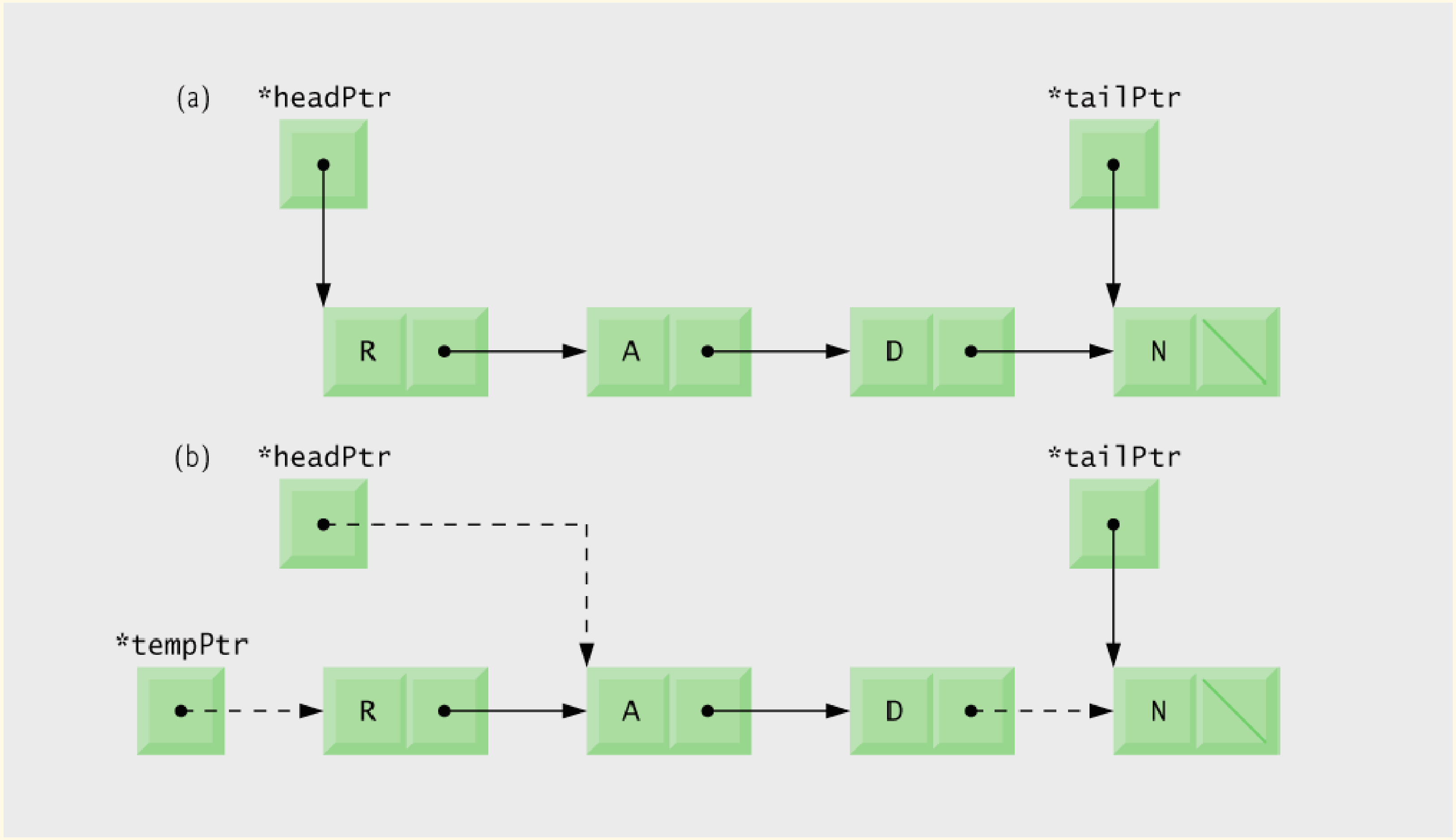


Fig 12.13 Processing a Queue

```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 */
}; /* end structure queueNode */

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 )
{
   QueueNodePtr headPtr = NULL; /* initialize headPtr */
   QueueNodePtr tailPtr = NULL; /* initialize tailPtr */
   int choice;                  /* user's menu choice */
   char item;                   /* char input by user */
```

Each node in the queue contains a data element and a pointer to the next node

Note that unlike linked lists and stacks, queues keep track of the tail node as well as the head

Fig 12.13 Processing a Queue

```

   instructions(); /* display the menu */
   printf( "? " );
   scanf( "%d", &choice );

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

      switch( choice ) {

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

         /* dequeue value */
         case 2:

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

            printQueue( headPtr );
            break;
```

Fig 12.13 Processing a Queue

```c

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;

      } /* end switch */

      printf( "? " );
      scanf( "%d", &choice );
   } /* end while */

   printf( "End of run.\n" );

   return 0; /* indicates successful termination */

} /* end main */

/* 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" );
} /* end function instructions */
```

Fig 12.13 Processing a Queue

```c

/* insert a node a queue tail */
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value )
{
   QueueNodePtr newPtr; /* pointer to new node */

   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;
      } /* end if */
      else {
         ( *tailPtr )->nextPtr = newPtr;
      } /* end else */

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

} /* end function enqueue */
```

To insert a node into the queue, memory must first be allocated for that node

Queue nodes are always inserted at the tail, so there is no need to search for the node's place

If the queue is empty, the inserted node becomes the new head in addition to the new tail

Inserted node becomes the new tail

Fig 12.13 Processing a Queue

```

/* remove node from queue head */
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
 {
  char value;           /* node value */
  QueueNodePtr tempPtr; /* temporary node pointer */

  value = ( *headPtr )->data;
  tempPtr = *headPtr;
  *headPtr = ( *headPtr )->nextPtr;

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

  free( tempPtr );

  return value;

} /* end function dequeue */

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

} /* end function isEmpty */
```

Queue nodes are always removed from the head, so there is no need to search for the node's place

Second node becomes the new head

If the removed node is the only node in the queue, it is the tail as well as the head of the queue, so **`tailPtr`** must be set to **`NULL`**

Free the memory of the removed node

Fig 12.13 Processing a Queue

```

/* Print the queue */
void printQueue( QueueNodePtr currentPtr )
{

   /* if queue is empty */
   if ( currentPtr == NULL ) {
      printf( "Queue is empty.\n\n" );
   } /* end if */
   else {
      printf( "The queue is:\n" );

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

      printf( "NULL\n\n" );
   } /* end else */

} /* end function printQueue */
```

Fig 12.13 Processing a Queue

```
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
```

(continued on next slide…)

Fig 12.13 Processing a Queue

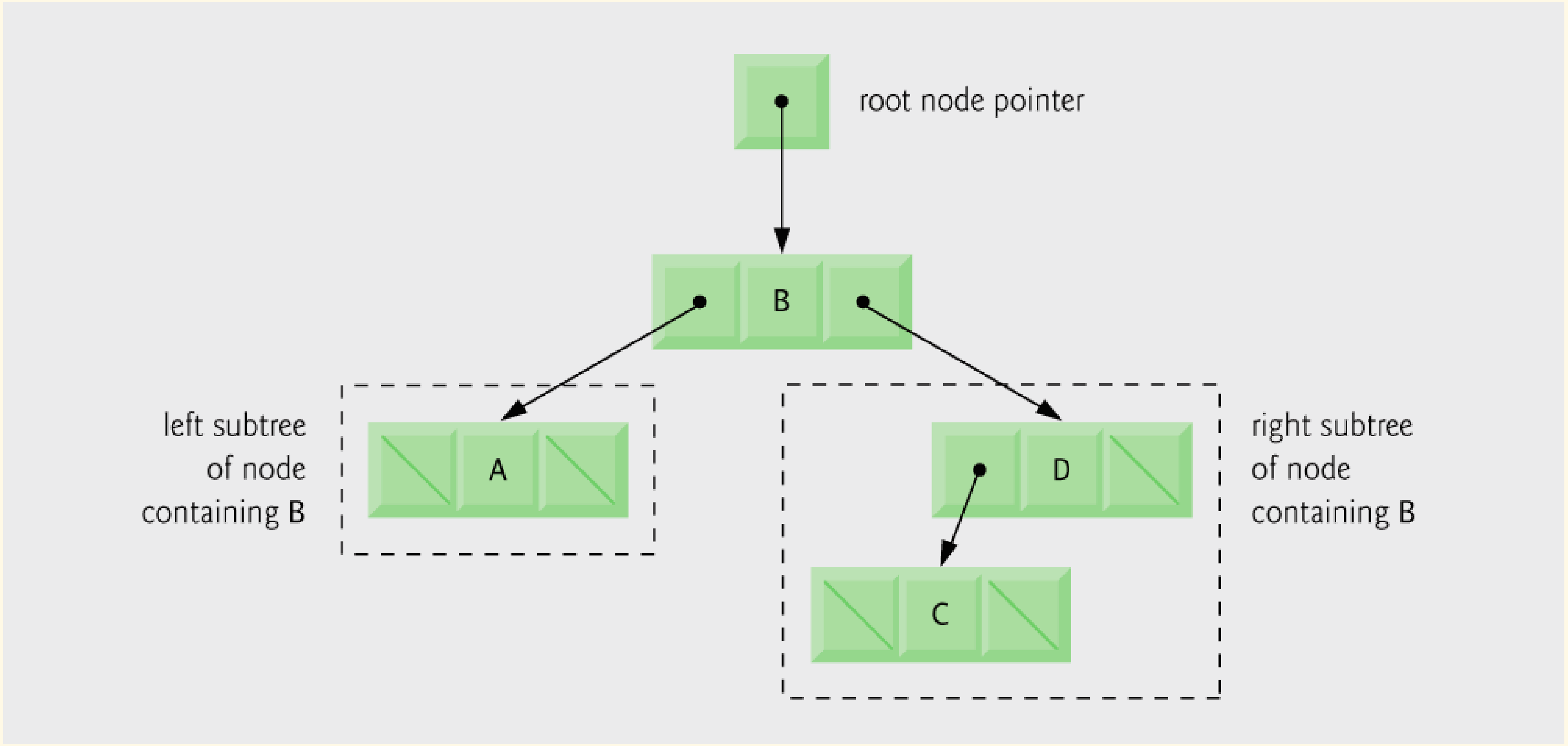
(continued from previous slide…)

```
? 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.
```

12.7 Trees

- Tree nodes contain **two** or more links.
  - All other data structures considered thus far have only contained one link.
- Binary trees
  - **All nodes contain two links.**
    - None, one, or both of which may be NULL
  - The **root node** is the first node in a tree.
  - Each link in the root node refers to a **child**.
  - A node with no children is called a **leaf node**.
  - A node can only be inserted as a leaf node in a binary search tree (i.e., a tree without duplicates.)

Fig. 12.17 Binary Tree Representation


root node pointer
B
A
D
C
left subtree of node containing B
right subtree of node containing B

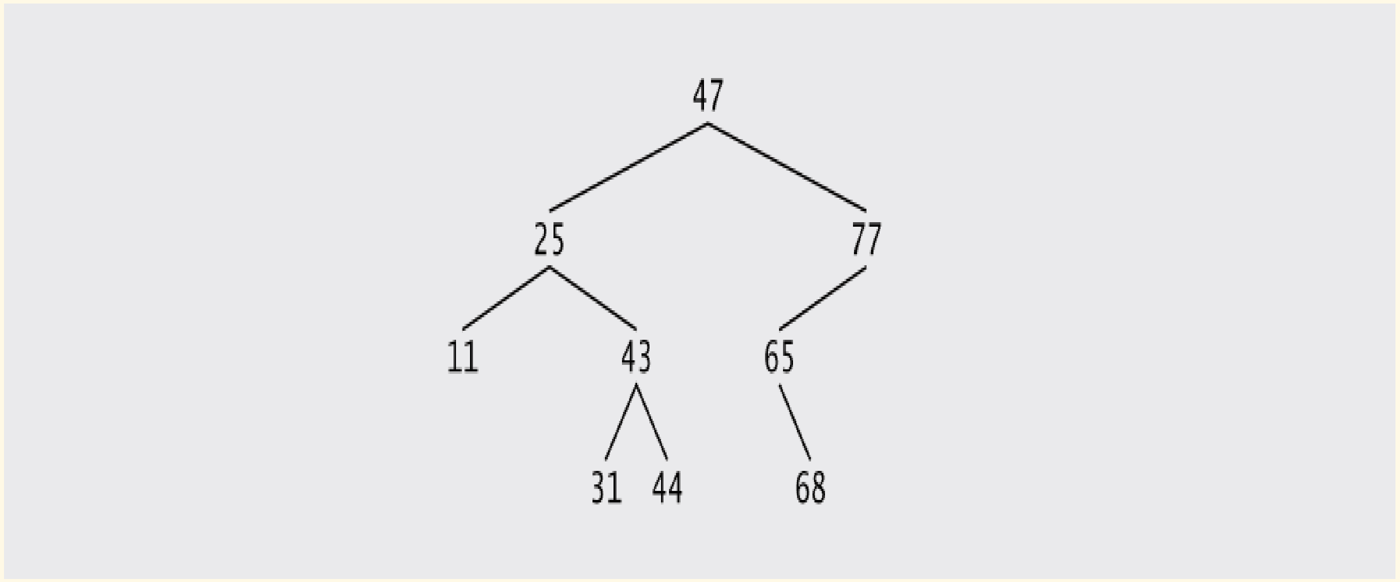
Common Programming Error 12.8

- Not setting to NULL the links in leaf nodes of a tree can lead to runtime errors.
- Remember, since tree traversals are normally recursive, this error can be 'deeply embedded'.

12.7 Trees

- Binary trees
  - The key value for nodes in the left subtree are less than the key value in the parent.
  - The key value for nodes in the right subtree are greater than the key value in the parent.
  - This data structure facilitates duplicate elimination!
  - Fast searches - for a balanced tree, maximum of $\log_2 n$ comparisons.

Fig. 12.18 Binary search tree



The Three Standard Tree Traversals

1. Inorder traversal:

- visit the nodes in **ascending order** by key value.

1.1 Traverse the left subtree with an inorder traversal

1.2 Visit the node (process the value in the node,i.e., print the node value).

1.3 Traverse the right subtree with an inorder traversal.

The Three Standard Tree Traversals

2. Preorder traversal:

2.1 Visit the node (process the value in the node, i.e., print the node value).

2.2 Traverse the left subtree with a preorder traversal.

2.3 Traverse the right subtree with a preorder traversal.

The Three Standard Tree Traversals

3. Postorder traversal:

3.1 Traverse the left subtree with a postorder traversal.

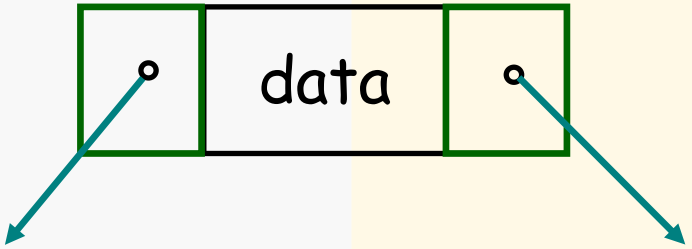
3.2 Traverse the right subtree with a postorder traversal.

3.3 Visit the node (process the value in the node, i.e., print the node value).

Tree Example

```c
/* Fig. 12.19: fig12_19.c
   Create a binary tree and traverse it
   preorder, inorder, and postorder */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* self-referential structure */
struct treeNode {
   struct treeNode *leftPtr;  /* pointer to left subtree */
   int data; /* node value */
   struct treeNode *rightPtr; /* pointer to right subtree */
}; /* end structure treeNode */

typedef struct treeNode TreeNode; /* synonym for struct treeNode */
typedef TreeNode *TreeNodePtr; /* synonym for TreeNode* */

/* prototypes */
void insertNode( TreeNodePtr *treePtr, int value );
void inOrder( TreeNodePtr treePtr );
void preOrder( TreeNodePtr treePtr );
void postOrder( TreeNodePtr treePtr );

/* function main begins program execution */
int main( void )
{
   int i; /* counter to loop from 1-10 */
   int item; /* variable to hold random values */
   TreeNodePtr rootPtr = NULL; /* tree initially empty */

```

Each node in the tree contains a data element and a pointer to the left and right child nodes



Tree Example

```
   srand( time( NULL ) );
   printf( "The numbers being placed in the tree are:\n" );

   /* insert random values between 0 and 14 in the tree */
   for ( i = 1; i <= 10; i++ ) {
      item = rand() % 15;
      printf( "%3d", item );
      insertNode( &rootPtr, item );
   } /* end for */

   /* traverse the tree preOrder */
   printf( "\n\nThe preOrder traversal is:\n" );
   preOrder( rootPtr );

   /* traverse the tree inOrder */
   printf( "\n\nThe inOrder traversal is:\n" );
   inOrder( rootPtr );

   /* traverse the tree postOrder */
   printf( "\n\nThe postOrder traversal is:\n" );
   postOrder( rootPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

Tree Example

```
/* insert node into tree */
void insertNode( TreeNodePtr *treePtr, int value )
{

   /* if tree is empty */
   if ( *treePtr == NULL ) {
      *treePtr = malloc( sizeof( TreeNode ) );

      /* if memory was allocated then assign data */
      if ( *treePtr != NULL ) {
         ( *treePtr )->data = value;
         ( *treePtr )->leftPtr = NULL;
         ( *treePtr )->rightPtr = NULL;
      } /* end if */
      else {
         printf( "%d not inserted. No memory available.\n", value );
      } /* end else */

   } /* end if */
   else { /* tree is not empty */

      /* data to insert is less than data in current node */
      if ( value < ( *treePtr )->data ) {
         insertNode( &( ( *treePtr )->leftPtr ), value );
      } /* end if */
```

To insert a node into the tree, memory must first be allocated for that node

If the inserted node's data is less than the current node's, the program will attempt to insert the node at the current node's left child.

Tree Example

```c

      /* data to insert is greater than data in current node */
      else if ( value > ( *treePtr )->data ) {
         insertNode( &( ( *treePtr )->rightPtr ), value );
      } /* end else if */
      else { /* duplicate data value ignored */
         printf( "dup" );
      } /* end else */

   } /* end else */

} /* end function insertNode */

/* begin inorder traversal of tree */
void inOrder( TreeNodePtr treePtr )
{

   /* if tree is not empty then traverse */
   if ( treePtr != NULL ) {
      inOrder( treePtr->leftPtr );
      printf( "%3d", treePtr->data );
      inOrder( treePtr->rightPtr );
   } /* end if */

} /* end function inOrder */

/* begin preorder traversal of tree */
void preOrder( TreeNodePtr treePtr )
{

```

If the inserted node's data is greater than the current node's, the program will attempt to insert the node at the current node's right child.

The inorder traversal calls an inorder traversal on the node's left child, then prints the node itself, then calls an inorder traversal on the right child.

Tree Example

```c
   /* if tree is not empty then traverse */
   if ( treePtr != NULL ) {
      printf( "%3d", treePtr->data );
      preOrder( treePtr->leftPtr );
      preOrder( treePtr->rightPtr );
   } /* end if */

} /* end function preOrder */

/* begin postorder traversal of tree */
void postOrder( TreeNodePtr treePtr )
{

   /* if tree is not empty then traverse */
   if ( treePtr != NULL ) {
      postOrder( treePtr->leftPtr );
      postOrder( treePtr->rightPtr );
      printf( "%3d", treePtr->data );
   } /* end if */

} /* end function postOrder */
```

The preorder traversal prints the node itself, then calls a preorder traversal on the node’s left child, then calls a preorder traversal on the right child.

The postorder traversal calls an postorder traversal on the node’s left child, then calls an postorder traversal on the right child, then prints the node itself.

12.5 Stacks

- Stack
  - New nodes are added and removed only at the top of the stack.
  - The classical analogy is a dishes stacker found in restaurants.
  - Last-in, first-out (LIFO) devices.
  - The bottom of stack is indicated by a link member to NULL.
  - Essentially, a constrained version of a linked list.
  - Stacks are important in computer languages because it is the data structure for calling and returning from function or subroutine calls.

12.5 Stacks

- Stack operations
- push
  - Adds a new node to the top of the stack.
- pop
  - Removes a node from the top of the stack.
  - Stores the popped value.
  - Returns true if pop was successful.

Fig. 12.10 Push Operation

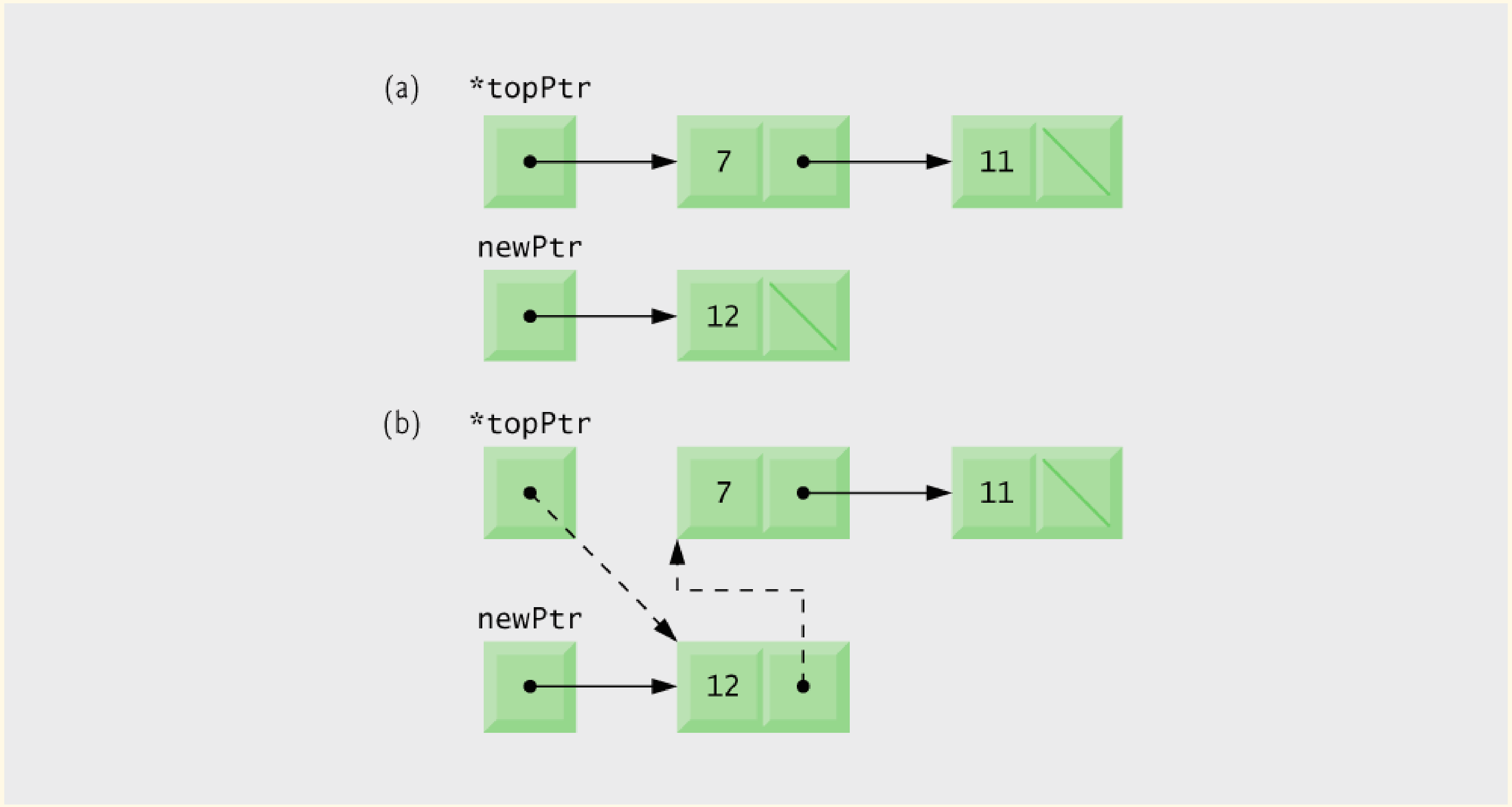
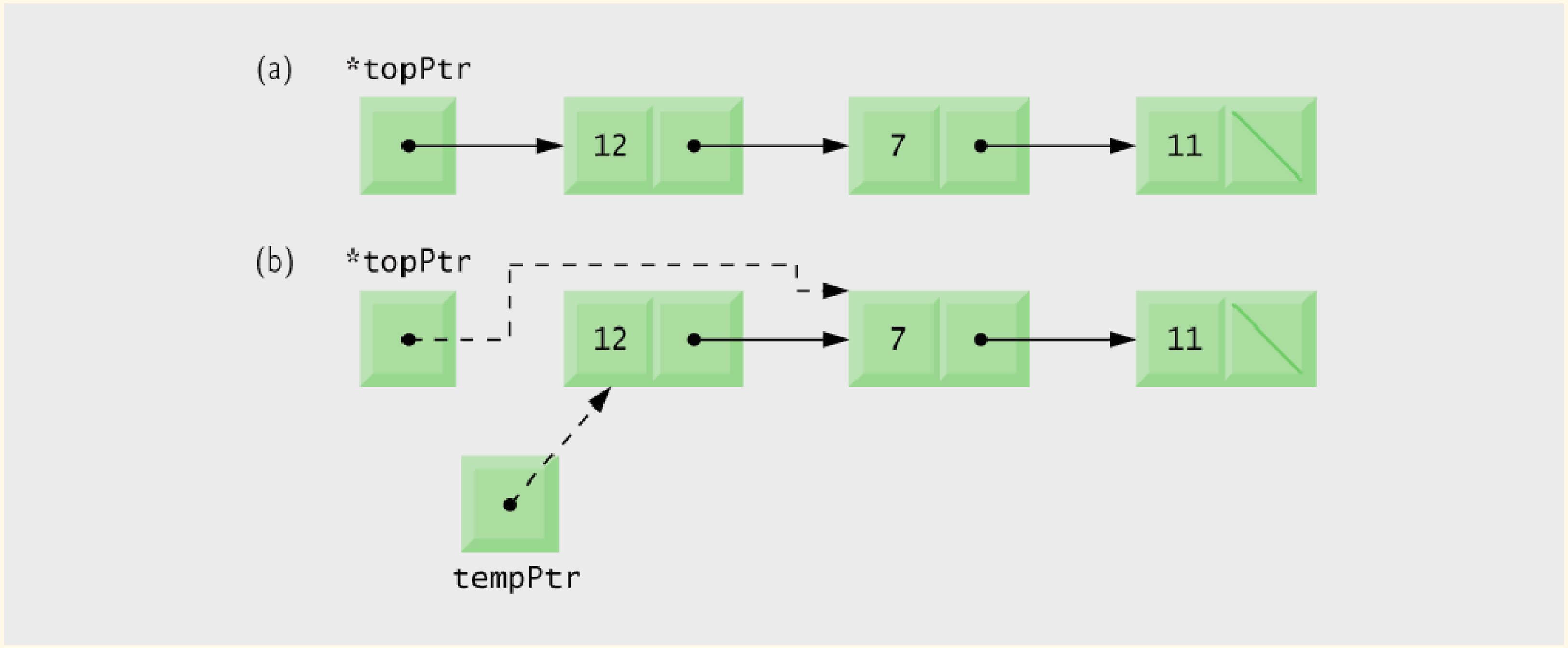


Fig. 12.10 Pop Operation



A Stack Example

```c
/* Fig. 12.8: fig12_08.c
   dynamic 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 */
}; /* end structure stackNode */

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 );

/* function main begins program execution */
int main( void )
{
   StackNodePtr stackPtr = NULL; /* points to stack top */
   int choice; /* user's menu choice */
   int value;  /* int input by user */

   instructions(); /* display the menu */
   printf( "? " );
```

Each node in the stack contains a data element and a pointer to the next node

A Stack Example

```
   scanf( "%d", &choice );

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

      switch ( choice ) {

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

         /* pop value off stack */
         case 2:

            /* if stack is not empty */
            if ( !isEmpty( stackPtr ) ) {
               printf( "The popped value is %d.\n", pop( &stackPtr ) );
            } /* end if */

            printStack( stackPtr );
            break;

         default:
            printf( "Invalid choice.\n\n" );
            instructions();
            break;
```

A Stack Example

```c

      } /* end switch */

      printf( "? " );
      scanf( "%d", &choice );
   } /* end while */

   printf( "End of run.\n" );

   return 0; /* indicates successful termination */

} /* end main */

/* display program instructions to user */
void instructions( void )
{
   printf( "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\n" );
} /* end function instructions */

/* Insert a node at the stack top */
void push( StackNodePtr *topPtr, int info )
{
   StackNodePtr newPtr; /* pointer to new node */

   newPtr = malloc( sizeof( StackNode ) );

```

To insert a node into the stack, memory must first be allocated for that node

A Stack Example

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

} /* end function push */

/* Remove a node from the stack top */
int pop( StackNodePtr *topPtr )
{
   StackNodePtr tempPtr; /* temporary node pointer */
   int popValue; /* node value */

   tempPtr = *topPtr;
   popValue = ( *topPtr )->data;
   *topPtr = ( *topPtr )->nextPtr;
   free( tempPtr );

   return popValue;

} /* end function pop */

```

Stack nodes are always inserted at the top, so there is no need to search for the node’s place

Inserted node becomes the new top

Stack nodes are always removed from the top, so there is no need to search for the node’s place

Second node becomes the new top

Free the memory of the popped node

A Stack Example

```
 /* Print the stack */
 void printStack( StackNodePtr currentPtr )
{

  /* if stack is empty */
  if ( currentPtr == NULL ) {
     printf( "The stack is empty.\n\n" );
  } /* end if */
  else {
     printf( "The stack is:\n" );

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

     printf( "NULL\n\n" );
  } /* end else */

} /* end function printList */
} /* end function printList */

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

} /* end function isEmpty */
```

A Stack Example

```
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


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
```

(continued on next slide…)

A Stack Example

(continued from previous slide…)

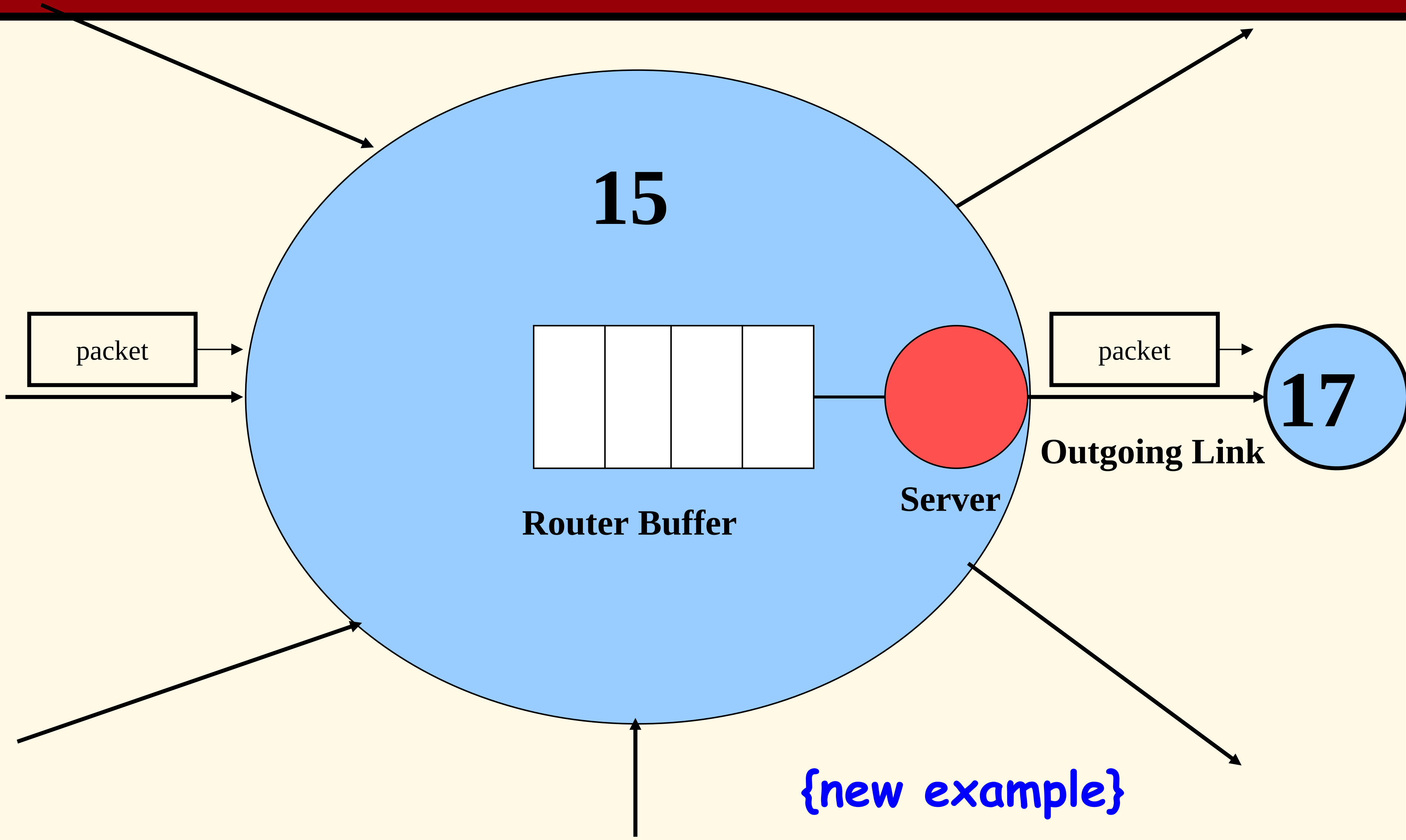
```
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 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.
```

Hashing

- Two classic examples of the use of hashing are:
- {old example}
- Building a symbol table for a compiler, whereby a symbol table is similar to a dictionary, but with a set of ***name-attribute*** pairs.
- Standard operations on any symbol table are:

1. Determine if the **name** is in the table.
2. Retrieve the attributes of that name.
3. Modify the attributes of that name.
4. Insert a new name and its attributes in the table.

Tracking Flows in a Router Node



{new example}

Static Hashing

- In static hashing, identifiers (names)
- Are stored in a fixed size table called a hash table.
- A hash function f(x) is used to determine the location of an identifier x in the table.
- The hash table is stored into sequential memory locations that are partitioned into b buckets. Each bucket has s slots.

Hash Table

Hash Function

- A good hash function is easy to compute and minimizes bucket collisions (i.e., it should be unbiased).
- A good hash function **hashes x** such that **x** has an equal chance of hashing into any of the **b** buckets.Namely, the hash function should **uniformly distribute** the symbols into the **b** buckets.
- Examples of hash function are: mid-square, division f(x) = x % M and folding.

Hash Table Details

- We need to initialize table where all slots are empty.

```
Void init_table (element ht[])
{
  int i;
  for (i =0; I < TABLE-SIZE; i++)
      ht[i].key[0] = NULL;
}
```

Hash Table with Linear Probing

There are four possibilities after hashing **x** into a table bucket:

1. The bucket contains **x**. {x is in the table already}
2. The bucket is empty.
3. The bucket contains a nonempty entry other than **x**. {a bucket collision}. Here we either examine next slot or the next bucket.
4. We have exhausted all table memory and have wrapped around to the 'home bucket'.

Hash Table with Linear Probing

- After running for awhile identifiers will tend to cluster and coalesce. Note, now the search time is likely to grow with each new addition to the symbol table.
- Result = Bad Performance.
- This is the motivation for implementing hash buckets using chaining (each bucket is implemented as a linked list with a header node).

Example of Chain insert into a hash table with a BAD hash function

[0] -> add -> asp -> attitude

[1] -> NULL

[2] -> cat -> char -> cosine -> czar

…

[9] -> jack -> jumbo

[10] -> king -> kinicki

…

[25] -> zac -> zebro -zits

More Advanced Hashing

- Dynamic Hashing
- Double Hashing

Summary of Data Structures

- Connected queuing computer systems with queuing data structures. Basic operations: enqueue and dequeue.
- Introduced tree terminology and structures
  - Binary Trees
  - Tree Traversals
    - Inorder, preorder, postorder
- Stacks
    - A Stack Example
- A quick look at Hashing including:
  - Static Hashing
  - Hash functions
  - Linear probing
  - Chaining (linked lists)