Chapter 12

Data Structures

Data Structures

Outline

- **Introduction**
- **Self-Referential Structures**
- **Dynamic Memory Allocation**
- **Linked Lists**

Introduction

- Dynamic data structures
  - Data structures are organized structs that grow and shrink during execution

- Important:
  - Since data structures are stored in main memory (RAM), they will be lost when the program ends.
  - If a data structure needs to be permanent, it should be saved into a file on hard disk.

Data Structures and Related Algorithms

•A data structure is a specific implementation of an Abstract Data Type.
• Most data structures have associated algorithms to perform operations, such as search, insert, delete that maintain the properties of the data structure.
• The C++ language contains built-in data structure libraries called Standard Template Library (STL).

- Data Structures
  - Arrays
  - Linked Lists
  - Stacks
  - Queues
  - Trees
  - Graphs

- Related Algorithms
  - Insert / Delete
  - Search
  - Sort
  - Traverse
  - Shortest path
  - etc.

Some Types of Data Structures

- **Linked list**
  - Allow insertions and removals anywhere in list
- **Stack (LIFO)**
  - Allow insertions and removals only at top of stack
- **Queue (FIFO)**
  - Allow insertions at the end and removals from the front of queue
- **Tree**
  - Insertions/removals are made in an ordered way in tree
  - High-speed searching and sorting of data

Implementation of Data Structures

- Array is the simplest but ineffective data structure.
- ***Linked List is the most fundemantal data structure.***
- All other data structures can be implemented by using either Arrays or Linked Lists.
- **NOTE:** We will study only Linked Lists in BIL105E course, because other types will be covered extensively in the Data Structures course.

Operations in Data Structures

- The followings are the most important operations:
  - **ADDING:** Add a new struct node
  - **DELETING:** Delete an existing struct node

- Other operations:
  - Print a data structure
  - Copy a data structure
  - Reverse a data structure
  - etc.

Self-Referential Structures

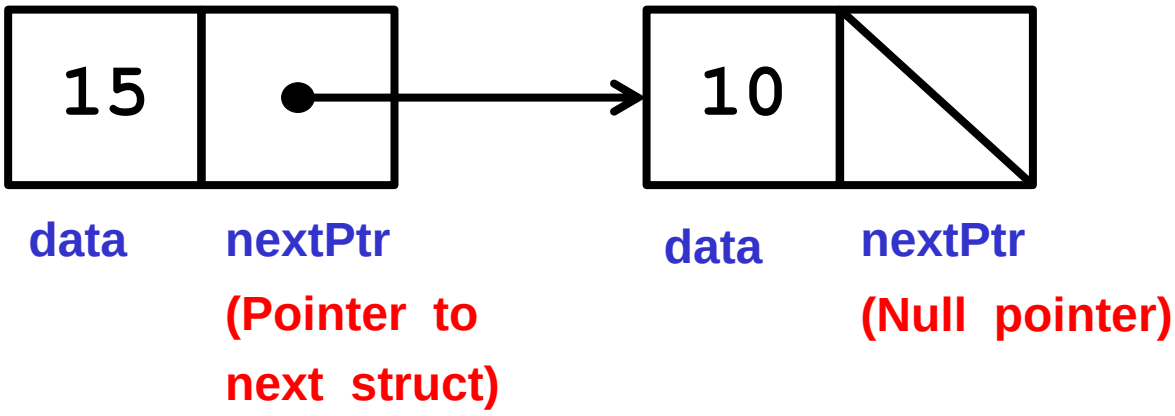
- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Can be linked together to form useful data structures such as lists, queues, stacks and trees
  - Terminated with a `NULL` pointer

```
struct node
{
    int data;
    struct node * nextPtr;
 }
```

- `nextPtr`
  - Points to an object of type node
  - Referred to as a link
    - Ties one `node` to another **`node`** (similar to a chain)

Dynamic Memory Allocation

Figure 12.1 Two self-referential structures linked together



Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - If no memory available, returns `NULL`
  - Example

```
newPtr = malloc( sizeof( struct node ) );
```

- `free`
  - Deallocates memory allocated by `malloc`
  - Takes a pointer as an argument

```
free ( newPtr );
```

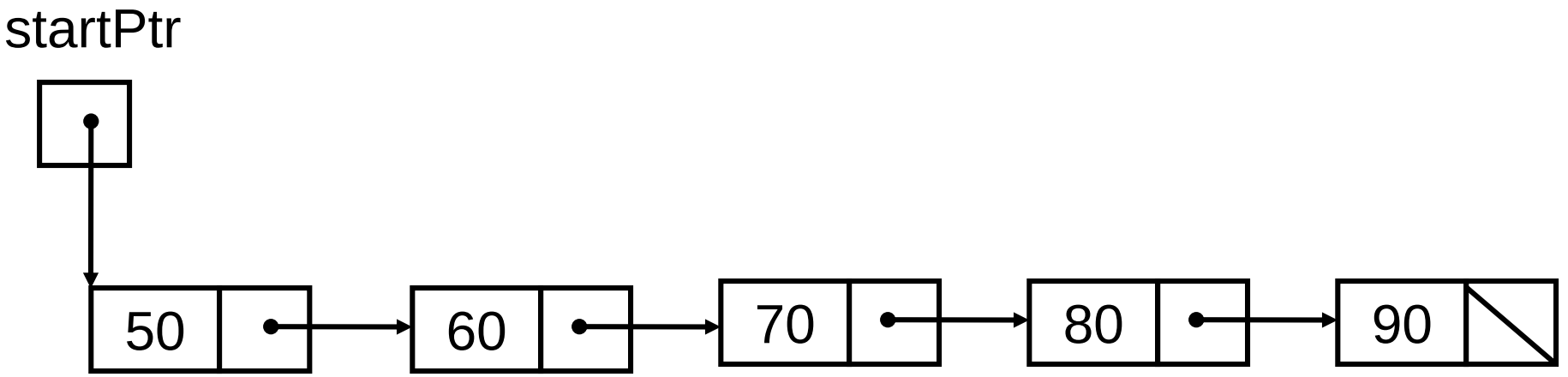
Linked Lists

Dynamic implementation of Linked List

- Dynamic Implementation (dynamic memory allocation)
- Memory usage is more economic.
- System manages free store (via malloc and free functions)
  - Easy to use
  - Storage is limited only by heap size
- Storage used proportional to number of nodes
  - Node locations are actual addresses
  - Links are pointers to nodes (Node *)
  - A special pointer to the first ***Node*** provides access to the first item in the list
  - A ***Node*** contains members ***data*** and ***nextPtr***

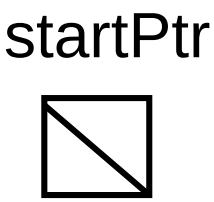
Dynamic implementation of Linked List

- Linked list is a chain of elements called nodes
- Each item is stored in a **node**
- Space is dynamically allocated (and returned) one **node** at a time
- Items are *not* stored in contiguous memory locations



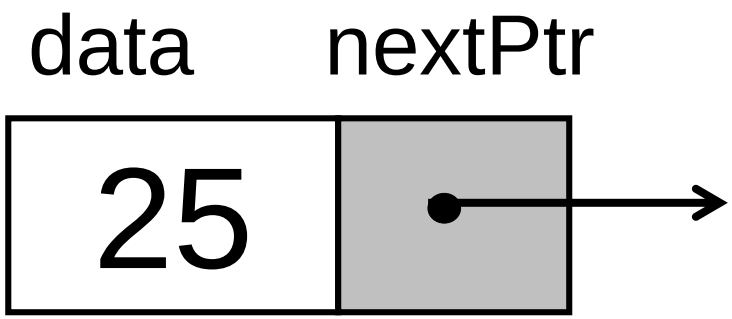
An empty linked list

- Has no items, so there are no nodes
- External pointer (startPtr) to first item has the value NULL

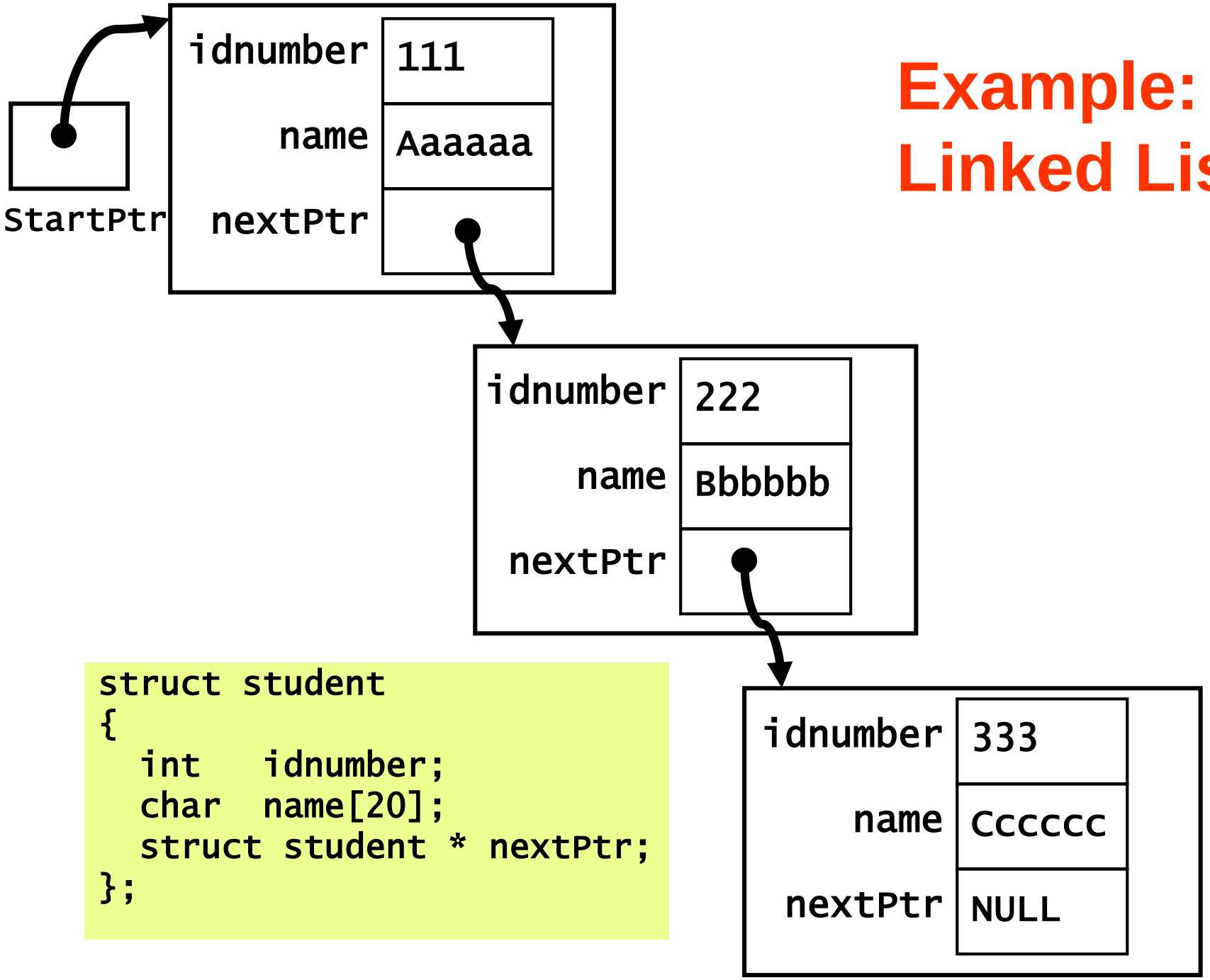


A node

- Each **node** contains a data part and a pointer to the next node
- Data part can contain many things such as student number, student name, etc.



Example: Linked List



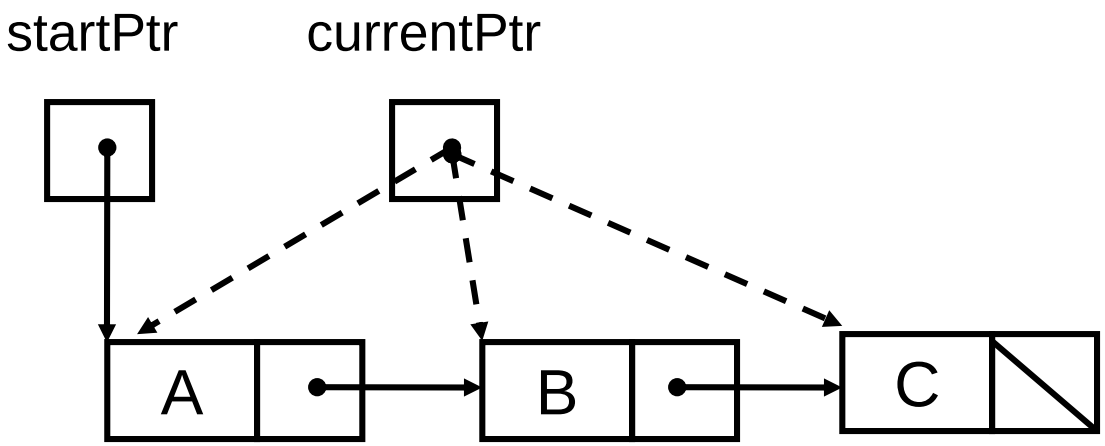

```
struct student
{
  int   idnumber;
  char  name[20];
  struct student * nextPtr;
};
```

Linked List Basic Operations

- **ADD:** Add an item at any position in the list (May involve searching)
- **DELETE:** Remove an item from the list at any position in the list (May involve searching)
- **TRAVERSE:** Go through the list accessing and processing (such as printing) the elements in order
- **EMPTY:** Check if list is empty

Code for Traversing

```
currentPtr = startPtr;
while ( currentPtr != NULL )
{
   printf("%d \n", currentPtr->data );
   currentPtr = currentPtr->nextPtr;
}
```



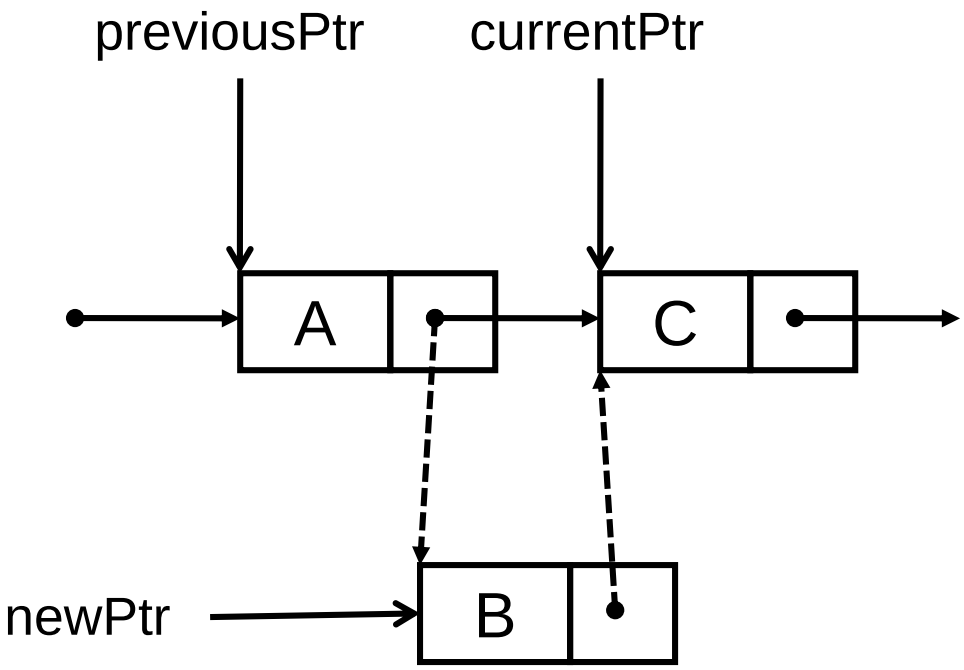
Code for Searching

```
// Same as Traversing
int found = FALSE;
currentPtr = startPtr;

// Search until target is found or
// the end of list is reached.
while ( currentPtr != NULL )
{
  if (currentPtr->data == target)
  {
      found =TRUE;
      break; // stop the while loop
  }
  currentPtr = currentPtr->nextPtr;
}
```

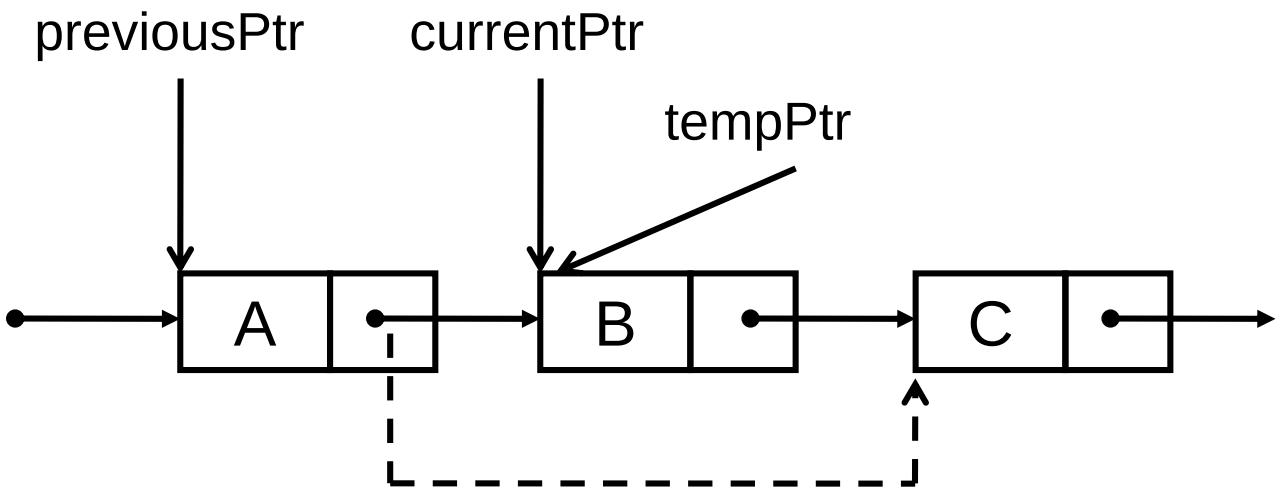
Code for Adding

```
newPtr = malloc( sizeof( ListNode ) ); // create node
newPtr->data = 'B'; // place value in node
previousPtr->nextPtr = newPtr;
newPtr->nextPtr = currentPtr;
```



Code for Deleting

```
tempPtr = currentPtr;
previousPtr->nextPtr = currentPtr->nextPtr;
free( tempPtr );
```



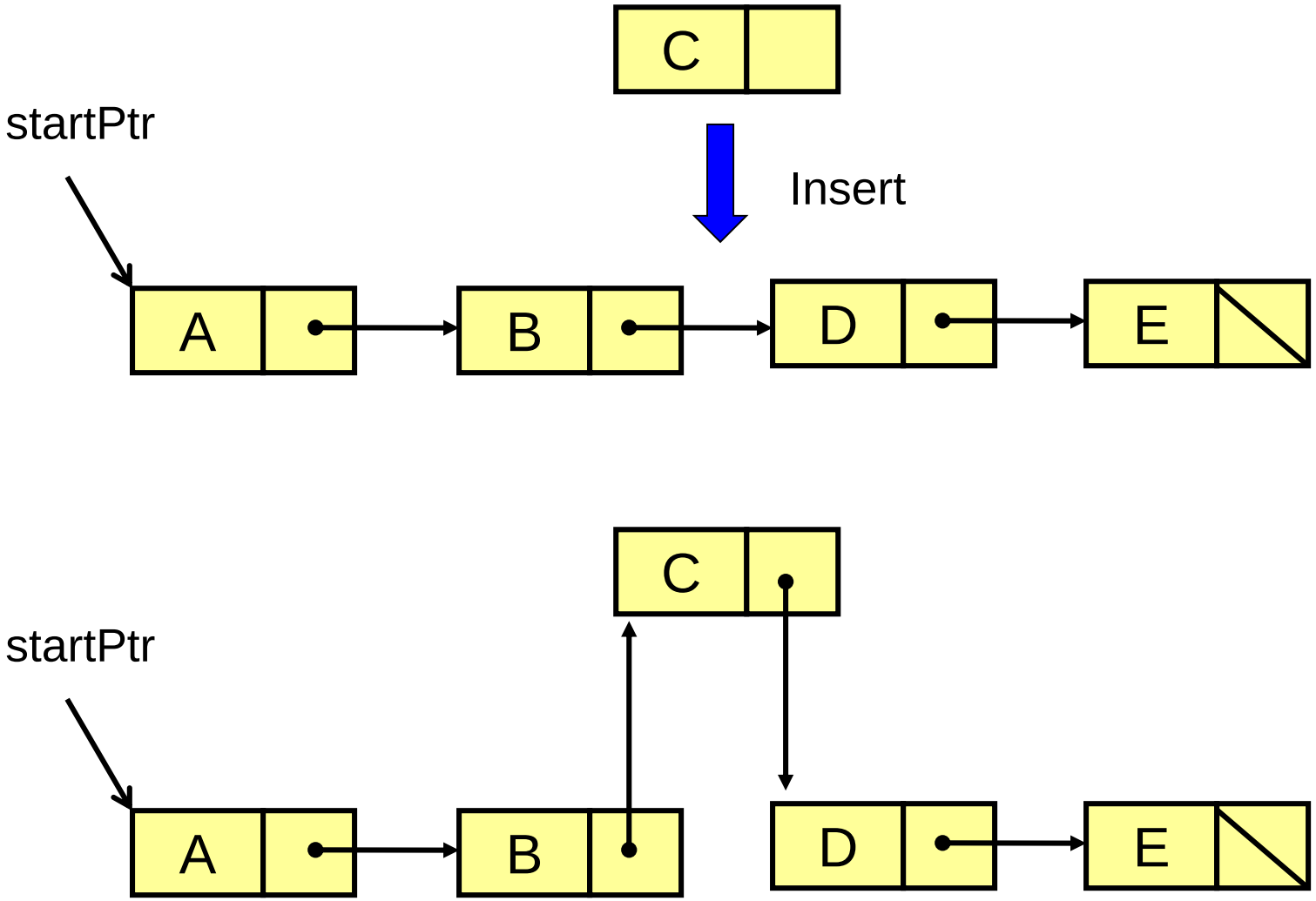
Adding an item

- Add operation requires two external pointers (previousPtr, currentPtr)
  - newPtr is placed between these two

- Operation takes the following steps:
  1. First, find the exact node location for insertion
  2. Allocate space for a new node (using malloc)
  3. Store the data item in that node
  4. Link the new node into the existing linked list

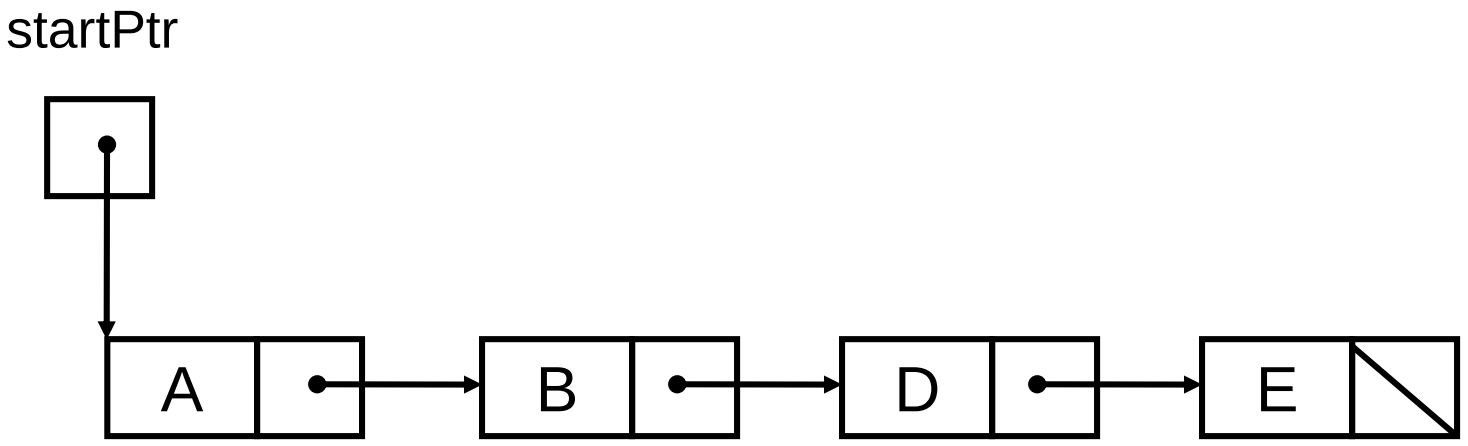
- Two cases for linking the node into the existing linked list
  - Adding an item at beginning of list
  - Adding an item at any other place, such as middle or at the end

Adding an item



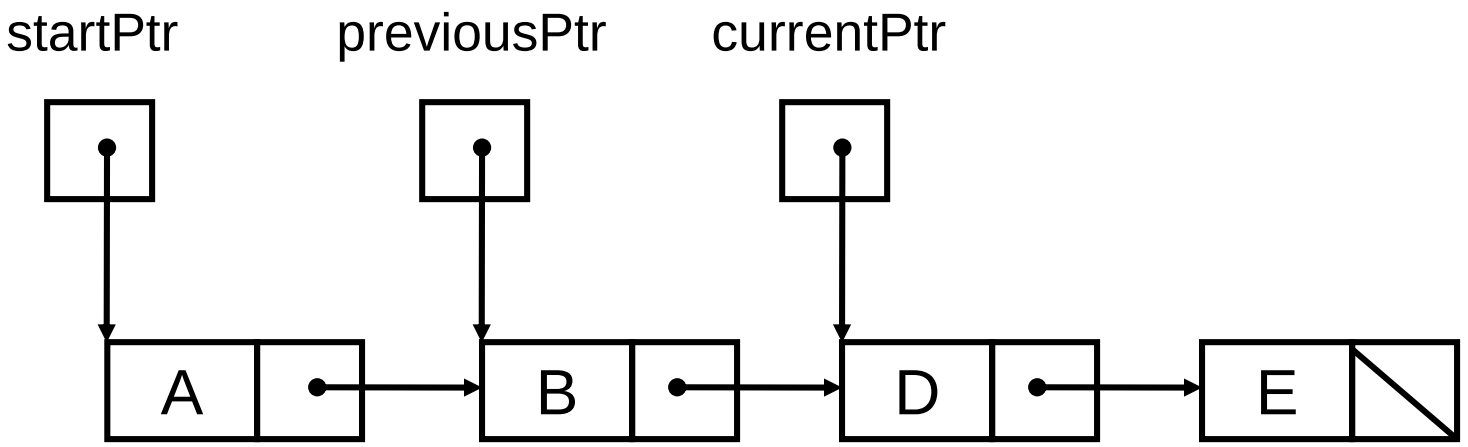
Adding an item (1)

Assume we want to add ‘C’ to the following list:



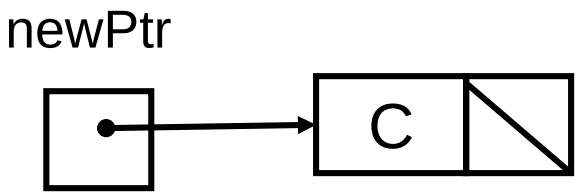
Adding an item (2)

Step 1) Find the exact location for insertion



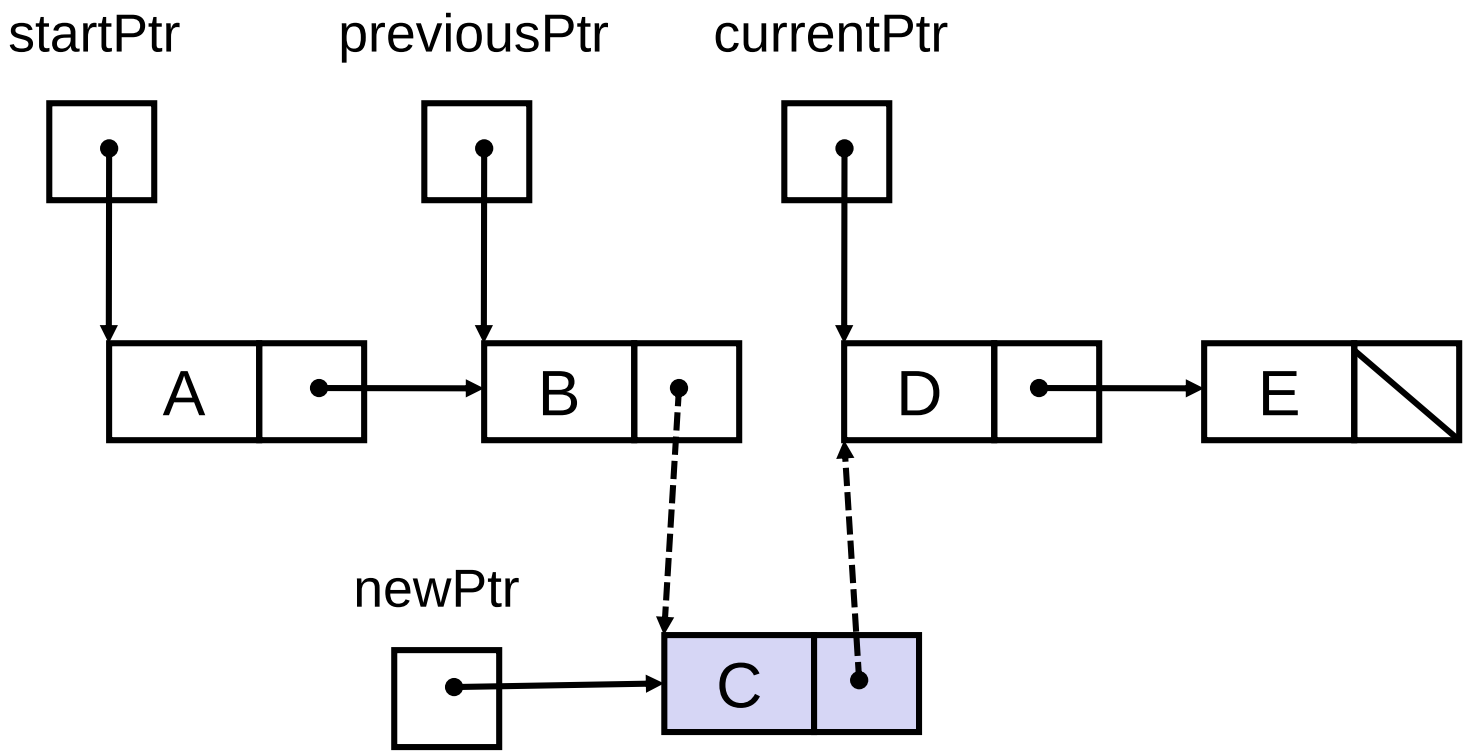
Adding an item (3)

Step 2 and 3) Allocate space for a new node and store the data item in that node.



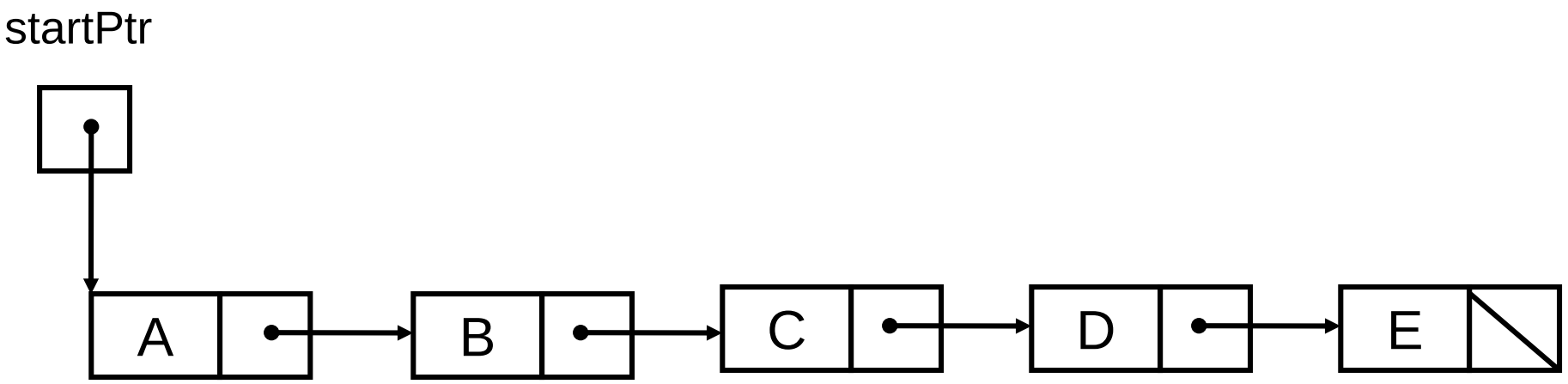
Adding an item (4)

Step 4) Link the new node between previousPtr and currentPtr.



Adding an item (5)

Final linked list:



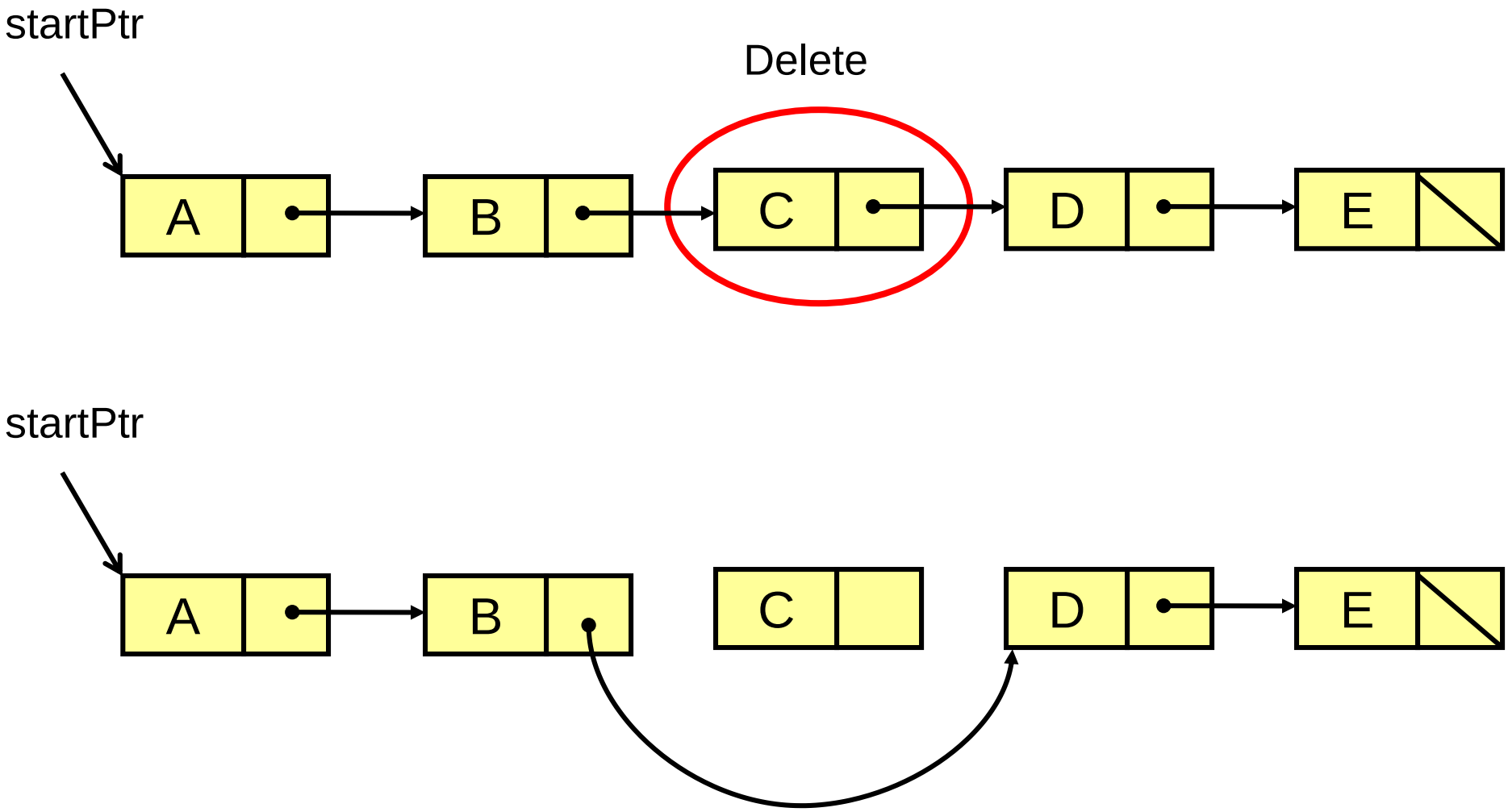
Deleting an item

- Delete operation requires two external pointers (previousPtr and currentPtr)
  - The node that is pointed by the currentPtr is deleted.

- Operation takes the following steps:
  1. First, find the node that will be deleted
  2. Change link value stored in previousPtr ->nextPtr
  3. De-allocate node containing element being deleted to the heap

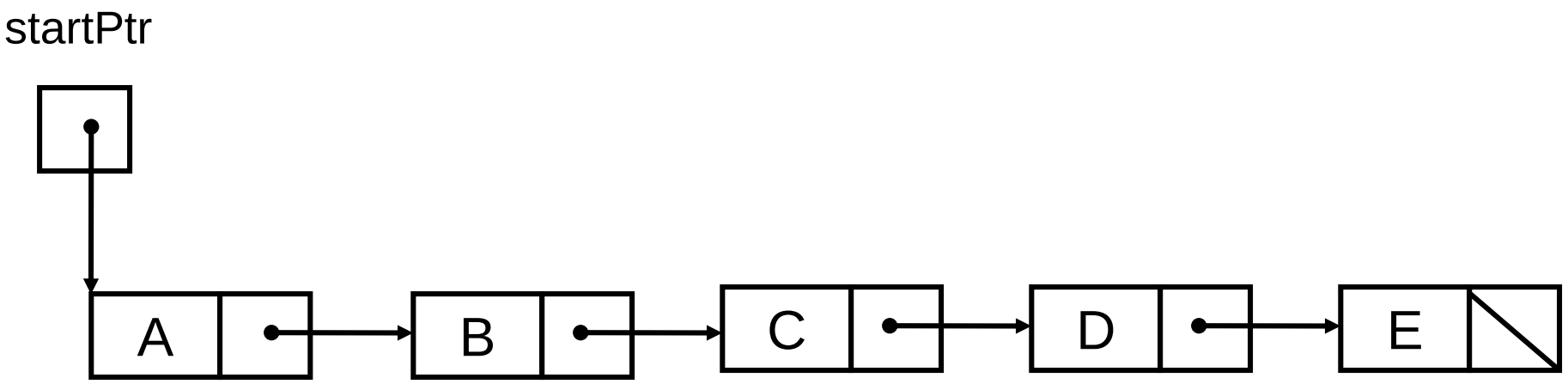
- Deleting first node is a special case
    - StartPtr must be changed

Deleting an item



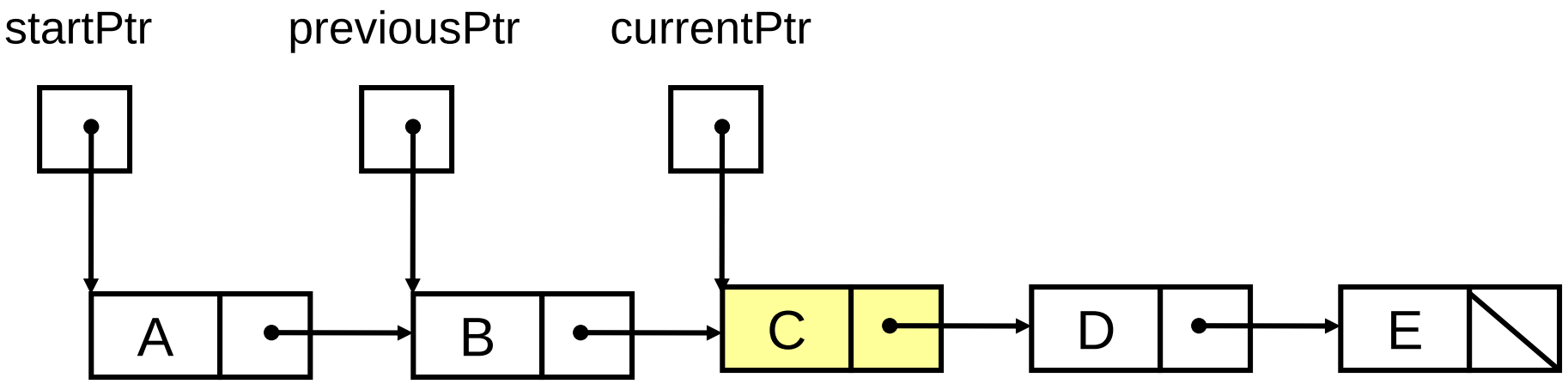
Deleting an item (1)

Assume we want to delete ‘C’ from the following list:



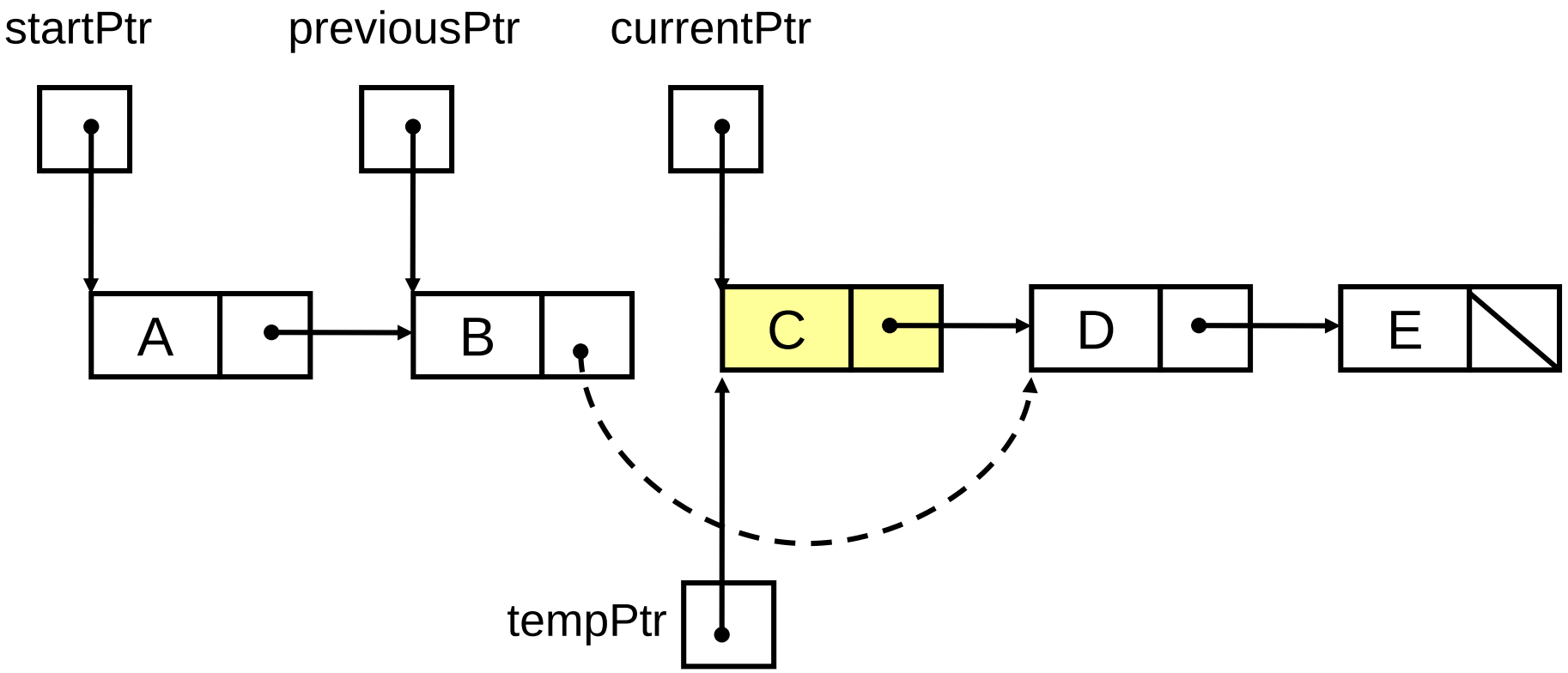
Deleting an item (2)

Step 1) Find the node that will be deleted.



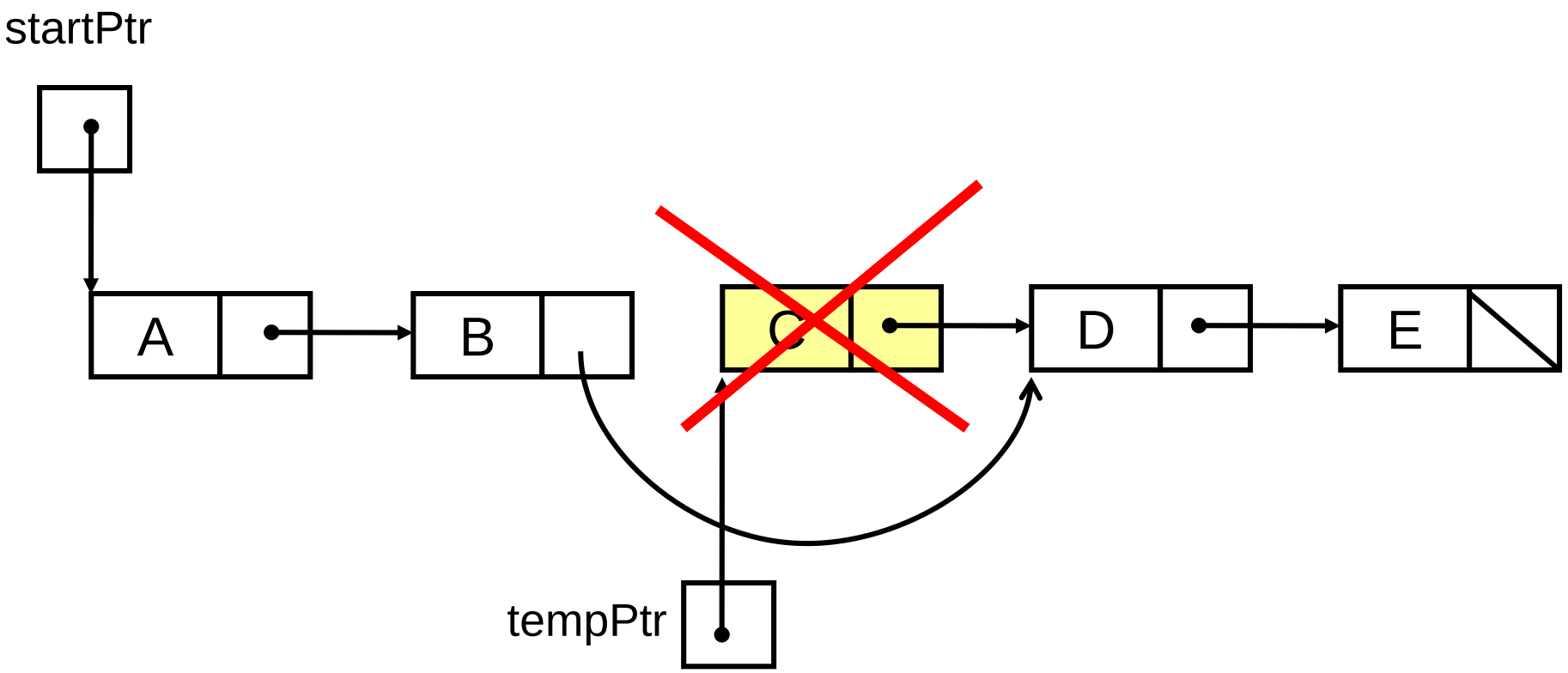
Deleting an item (3)

Step 2) Change link value stored in previousPtr ->nextPtr



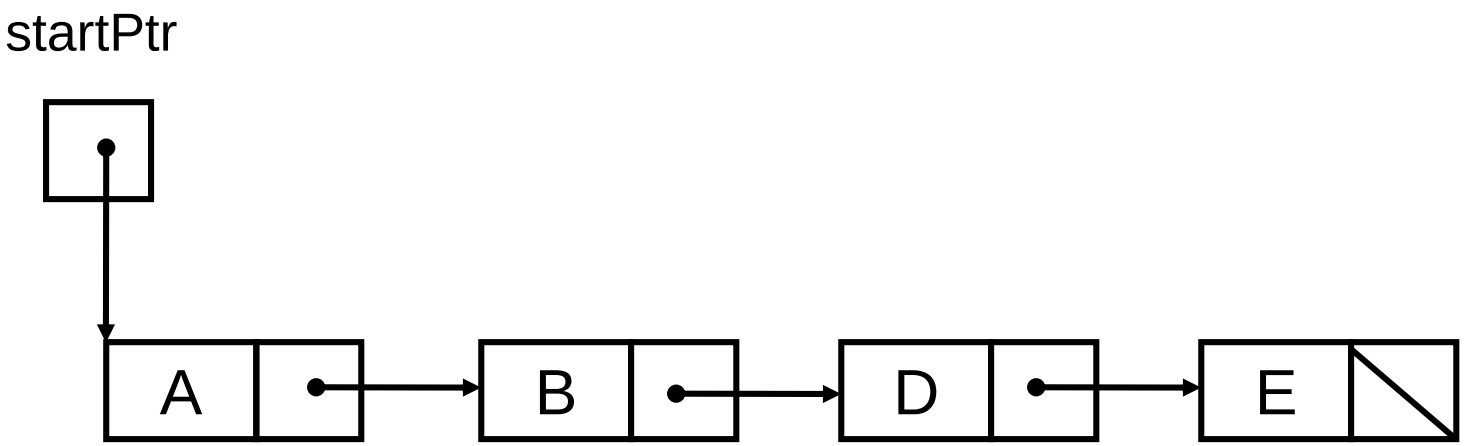
Deleting an item (4)

Step 3) De-allocate node containing element being deleted to the heap



Deleting an item (5)

Final linked list:

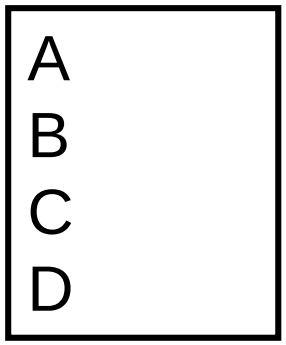


Saving and Restoring a Linked List to/from a File

- The following steps are used to store the list in a file.
  1) At the beginning of program, read all data from file and recreate the list by adding each item to the list one by one.
  2) Set a flag variable to TRUE, if user makes any modifications (add and/or delete) on the list.
  3) At the end of program, check the flag. If the flag is TRUE, then save all data items back to the file.

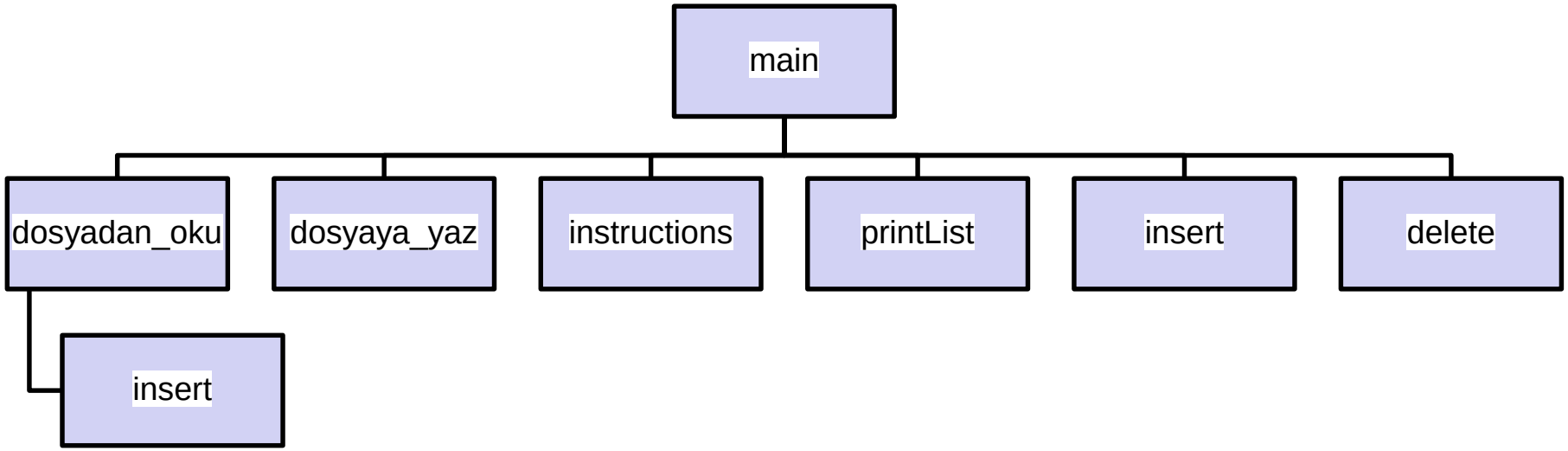
- **IMPORTANT: Do not save pointers to the file, save only the data.**
  - The pointers are real memory addresses which must be allocated dynamically at run-time.
  - Therefore, whenever you run the program they must be re-allocated dynamically.

Example content of a file:



EXAMPLE:
Linked List Program

Program Modules




```
// Function prototypes

void insert( ListNodePtr *sPtr, int value );
int delete( ListNodePtr *sPtr, int value );
void printList( ListNodePtr currentPtr );
void instructions( void );
void dosyadan_oku(ListNodePtr *sPtr);
void dosyaya_yaz(ListNodePtr currentPtr);
```

User Interface Instructions (Menu)

```
// display program instructions to user
void instructions( void )
{
   printf( "Enter your choice:\n"
      "   1) Insert an element into the list\n"
      "   2) Delete an element from the list\n"
      "   3) Print the list\n"
      "   4) Exit\n ? " );
}
```

Struct Definitions

```
// Program maintains an ordered linked list of integer numbers.
// Data are read from and written to a file.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TRUE 1
#define FALSE 0

// self-referential structure
struct Node
{
  int  data;
  struct Node *nextPtr;  // pointer to next node
};

typedef struct Node ListNode; // synonym for struct Node
typedef ListNode *ListNodePtr;  // synonym for ListNode*
```

main (1)

```
int main()
{
   ListNodePtr startPtr = NULL;  // initially there are no nodes
   int choice; // user's choice
   int data;  // data entered by user
   int modified_flag = FALSE;
   // Dosyadan okunan listede degisim tesbiti icin.

   char cevap;  // Dosyaya kaydetmek icin kullanici cevabi

   printf( "LINKED LIST PROGRAM EXAMPLE\n" );

   dosyadan_oku(&startPtr);

   printList( startPtr );

   instructions(); // display the menu
   scanf( "%d", &choice );
```

main (2)

```
// loop while user does not choose exit
   while ( choice != 4 ) {

      switch ( choice ) {

         case 1:
            printf( “Enter a number as data: " );
            scanf("%d", &data );
            insert( &startPtr, data );
            // insert new data in list

            modified_flag = TRUE;
            printList( startPtr );
            break;
```

main (3)

```
case 2:
            // if list is not empty
            if ( !isEmpty( startPtr ) ) {
               printf( "\nEnter data value to be deleted: " );
               scanf("%d", &data );
               // if name is found, remove it
               if ( delete( &startPtr, data ) ) {
                  // removed data's node
                  printf( "%d deleted.\n", data );
                  modified_flag = TRUE;
                  printList( startPtr );
               } // end if
               else
                  printf( "\n%d not found.\n\n", data );
            } // end if
            else
               printf( "\nList is empty.\n\n" );
            break;
```

main (4)

```
case 3:
            printList( startPtr );
            break;

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

      } // end switch

      instructions(); // display the menu
      scanf( "%d", &choice );
   } // end while
```

main (5)

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

 if (modified_flag == TRUE)
 {
      printf("Listede degisim tespit edildi.\n");
      printf("Dosyaya kaydetmek istermisiniz (E/H) ? ");

      fflush(stdin); //clear keyboard buffer
      cevap = getchar();

      if (cevap == 'e' || cevap == 'E')
         dosyaya_yaz(startPtr);
 }

} // end main
```

printList

```
// Print data items in the list
void printList( ListNodePtr currentPtr )
{  // if list is empty
   if ( currentPtr == NULL )
   {
      printf( "List is empty.\n" );
   }
   else {
      printf( "The list is:\n" );
      // while not the end of the list
      while ( currentPtr != NULL )
  {
         printf( "%d --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      } // end while
      printf( "NULL\n" );
   } // end else

}  // end function
```

dosyadan_oku (1)

```
void dosyadan_oku(ListNodePtr *sPtr)
{
  FILE *fPtr;
  int data;  // data value read from file

  fPtr = fopen("veriler.txt","r");

  if (fPtr == NULL)
  {
     printf("veriler.txt dosyasi bulunamadi.\n");
     return;
  }

  printf("veriler.txt dosyasi bulundu.\n");
```

dosyadan_oku (2)

```
 // Read first record
  fscanf(fPtr, "%d", &data);

  while(!feof(fPtr))
  {
     insert( sPtr, data ); // insert data in list
     fscanf(fPtr, "%d", &data); // Read next record
  }

  fclose(fPtr);

} // end of dosyadan_oku
```

dosyaya_yaz

```
void dosyaya_yaz(ListNodePtr currentPtr)
{
  FILE *fPtr;
  fPtr = fopen("veriler.txt","w");
  if (fPtr == NULL)
  {
     printf("veriler.txt dosyasi olusturulamadi.\n");
     return;
  }

  // while not the end of the list
  while ( currentPtr != NULL )
  {
    fprintf(fPtr, "%d\n", currentPtr->data);
    currentPtr = currentPtr->nextPtr;
  } // end while

  fclose(fPtr);
} // end of dosyaya_yaz
```

insert (1)

```
/* Insert a new value into the list in sorted order */
void insert( ListNodePtr *sPtr, int value )
{
   ListNodePtr newPtr;      // pointer to new node
   ListNodePtr previousPtr; // pointer to previous node
   ListNodePtr currentPtr;  // pointer to current node

   newPtr = malloc( sizeof( ListNode ) ); // create node

   if ( newPtr != NULL ) { // is space available
      newPtr->data = value; // place value in node
      newPtr->nextPtr = NULL;
     // node does not link to another node

      previousPtr = NULL;
      currentPtr = *sPtr;
```

insert (2)

```
// loop to find the correct location in the list
      while ( currentPtr != NULL && value > currentPtr->data )
      {
         // walk to next node
         previousPtr = currentPtr;
         currentPtr = currentPtr->nextPtr;
      } // end while

      // insert new node at beginning of list
      if ( previousPtr == NULL ) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
         // Starting node of list has been changed!

      } // end if
```

insert (3)

```
else {
         // insert new node between previousPtr and currentPtr
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      } // end else

   } // end if
   else {
      printf( “%s not inserted. No memory available.\n", value );
   } // end else

} // end function insert
```

delete (1)

```
// Delete a list element
int delete( ListNodePtr *sPtr, int value )
{
  ListNodePtr previousPtr; // pointer to previous node
  ListNodePtr currentPtr; //pointer to current node
  ListNodePtr tempPtr;    // temporary node pointer

   // delete first node if matches the value
   if ( value == ( *sPtr )->data )
   {
      tempPtr = *sPtr; // hold onto node being removed
      *sPtr = ( *sPtr )->nextPtr; // de-thread the node
      // Starting node of list has been changed!
      free( tempPtr ); // free the de-threaded node
      return value;
   } // end if
```

delete (2)

```
else {
      previousPtr = *sPtr;
      currentPtr = ( *sPtr )->nextPtr;

      // loop to find the correct location in the list
      while ( currentPtr != NULL && currentPtr->data != value  )
      {
         // walk to next node
         previousPtr = currentPtr;
         currentPtr = currentPtr->nextPtr;
      } // end while
```

delete (3)

```
 // delete node at currentPtr
      if ( currentPtr != NULL )
      {
         tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free( tempPtr );
         return TRUE; // 1
      } // end if

   } // end else

   return FALSE; // 0

} // end function delete
```

Program Run (1)

```
LINKED LIST PROGRAM EXAMPLE

veriler.txt dosyasi bulundu.

The list is:
20 --> 50 --> 80 --> NULL



Enter your choice:
   1) Insert an element into the list
   2) Delete an element from the list
   3) Print the list
   4) Exit
?
```

Program Run (2)

```
Enter your choice:
   1) Insert an element into the list
   2) Delete an element from the list
   3) Print the list
   4) Exit
? 2

Enter data value to be deleted: 30

30 not found.
```

Program Run (3)

```
Enter your choice:
   1) Insert an element into the list
   2) Delete an element from the list
   3) Print the list
   4) Exit
? 1

Enter a number as data: 40

The list is:
20 --> 40 --> 50 --> 80 --> NULL
```

Program Run (4)

```
Enter your choice:
   1) Insert an element into the list
   2) Delete an element from the list
   3) Print the list
   4) Exit
? 4

End of run.

Listede degisim tespit edildi.
Dosyaya kaydetmek istermisiniz (E/H) ? e
```