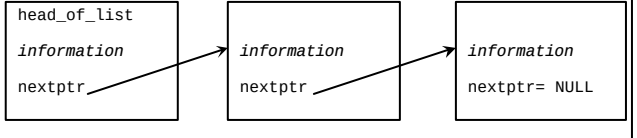
Linked List

Dynamic Data Structures

- Dynamic data structures
  - Data structures that grow and shrink during execution
- Linked lists
  - Allow insertions and removals anywhere

The linked list - a common use of structs which contain themselves

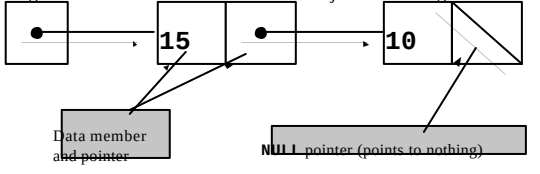
- Imagine we are reading lines from a file but don't know how many lines will be read.
- We need a structure which can extend itself.



This is known as a linked list. By passing the value of the head of the list to a function we can pass ALL the information.

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 (**`0`**)
- Diagram of two self-referential structure objects linked together



Self-Referential Classes

- Example

```
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**

Linked list pros & cons

- Pro:
  - Can easily add items to the head or middle of the list (tough with array) - good for ordered lists
  - We can make the list as big as we like
- Con:
  - Complicated for a beginner to program (don't underestimate this)
  - Hard to find nth element
  - Slightly larger since we store pointer and information

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- **free**
  - Deallocates memory allocated by **malloc**
  - Takes a pointer as an argument
  - **free (newPtr);**

Dynamic Memory Allocation

- **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

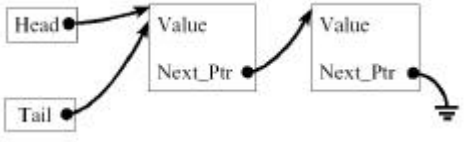
```
newPtr = malloc( sizeof( struct
  node ) );
```

Dynamic Linked Lists

- Dynamic Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to null to mark the list's end
- Use a dynamic linked list instead of a static array when
  - You have an unpredictable number of data elements
  - Your list needs to be sorted quickly

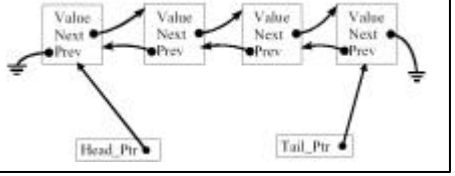
Types of Dynamic Linked Lists

- Singly linked list
  - Begins with a pointer to the first node
  - Terminates with a null pointer
  - Only traversed in one direction
- Circular, singly linked
  - Pointer in the last node points back to the first node

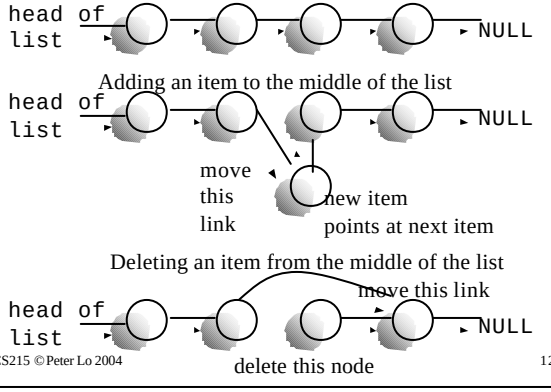


Types of Dynamic Linked Lists

- Doubly Linked List
  - Two " start pointers" – first element and last element
  - Each node has a forward pointer and a backward pointer
  - Allows traversals both forwards and backwards
- Circular, Doubly Linked List
  - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node



Linked List Operation

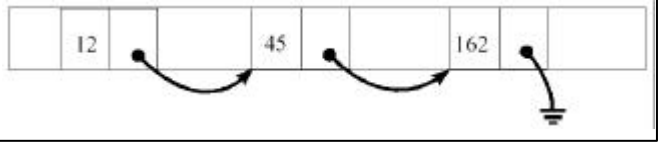


Examples

- In these examples, the following structure will be used extensively:

```
struct Node {
  int Value;
  struct Node *Next_Ptr;
};
```

- Example linked list:



Forming a Linked List

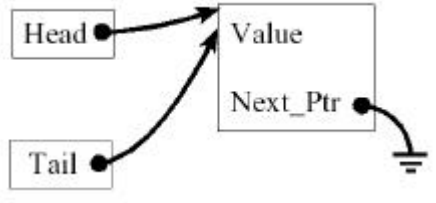
- Forward Growing (Adding to the End):
  - Use one pointer to refer to the "head" of the list.
  - Use a second pointer to refer to the "tail" of the list.
  - Add every new structure to Tail->Next_Ptr.
- Reverse Growing (Adding to the Beginning):
  - Use one pointer to refer to the "head" of the list.
  - Use a temporary pointer to refer to a new structure.
  - Set Temp_Ptr->Next_Ptr = Head_Ptr
  - Set Head = Temp_Ptr

Forward Growing

- The first thing we must do in either case is allocate the Head.
  - if (Head_Ptr == NULL) {
    - Head_Ptr = malloc (sizeof (struct Node));
    - assert(Head_Ptr != NULL);
    - Head_Ptr->Next_Ptr = NULL;
  - }

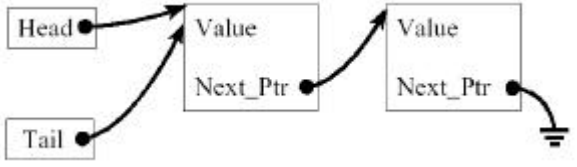
Forward Growing

- In the forward growing case, we then set the Tail to the Head.
  - Tail_Ptr = Head_Ptr;



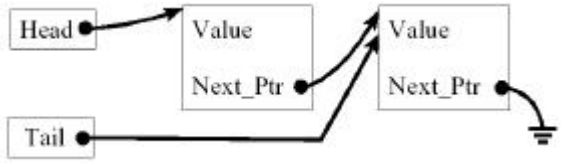
Forward Growing

- Tail->Next_Ptr = malloc(sizeof (struct Node));
- assert(Tail->Next_Ptr != NULL);
- Tail->Next_Ptr->Next_Ptr = NULL;



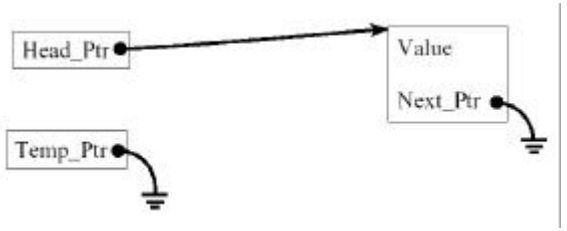
Forward Growing

- Tail = Tail->Next_Ptr;



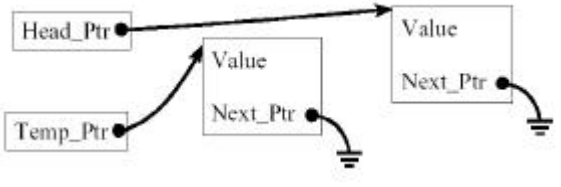
Reverse Growing

- In the reverse growing case, we then set the Temp pointer to NULL first.



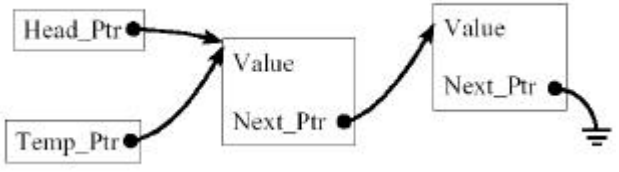
Reverse Growing

- Temp_Ptr = malloc (sizeof (struct Node));
- assert (Temp_Ptr != NULL);
- Temp_Ptr->Next_Ptr = NULL;



Reverse Growing

- Temp_Ptr->Next_Ptr = Head_Ptr;
- Head_Ptr = Temp_Ptr;



Reverse Growing

- Temp_Ptr->Next_Ptr = Head_Ptr;
- Head_Ptr = Temp_Ptr;

Example 9A

```
/* 9A: Operating and maintaining a dynamic linked list */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

struct listNode {   /* self-referential structure */
   char data;
   struct listNode *nextPtr;
};

typedef struct listNode ListNode;
typedef ListNode *ListNodePtr;

void insertnode( ListNodePtr *, char );
char deletenode( ListNodePtr *, char );
int isEmpty( ListNodePtr );
void printList( ListNodePtr );
void instructions( void );
```


Example 9A (cont')

```
int main()
{
   ListNodePtr startPtr = NULL;
   int choice;
   char item;

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

   while ( choice != 3 ) {

      switch ( choice ) {
         case 1:
            printf( "Enter a character: " );
            scanf( "\n%c", &item );
            insertnode(&startPtr, item );
            printList( startPtr );
            break;
```

Example 9A (cont')

```
        case 2:
          if ( !isEmpty( startPtr ) ) {
            printf( "Enter character to be deleted: " );
            scanf( "\n%c", &item );
            if ( deletenode ( &startPtr, item ) ) {
              printf( "%c deleted.\n", item );
              printList ( startPtr );
            }
            else
              printf( "%c not found\n\n", item );
          }
          else
            printf( "List is empty\n\n" );
          break;
        default:
          printf( "Invalid choice. \n\n" );
          instructions();
          break;
      }
      printf( "? " );
      scanf ( "%d", &choice );
    }
```


Example 9A (cont')

```
  printf( "End of run.\n" );
        getch();
  return 0;
}

/* Print the instructions */
void instructions( void )
{
  printf( "Enter your choice:\n"
          "  1 to insert an element into the list.\n"
          "  2 to delete an element from the list.\n"
          "  3 to end.\n" );
}
```


Example 9A (cont')

```
/* Insert a new value into the list in sorted order */
void insertnode ( ListNodePtr *sPtr, char value ) {
  ListNodePtr newPtr , previousPtr , currentPtr;
  (void *) newPtr = malloc ( sizeof( ListNode ) ); /* malloc returns pointer to void */
  if ( newPtr != NULL ) {     /* is space available */
    newPtr->data = value;
    newPtr->nextPtr = NULL;
    previousPtr = NULL;
    currentPtr = *sPtr;
    while ( currentPtr != NULL && value >currentPtr->data ) {
      previousPtr = currentPtr;        /* walk to ...  */
      currentPtr  = currentPtr->nextPtr;  /* ... next node */        }
    if ( previousPtr == NULL ) {
      newPtr->nextPtr = *sPtr;
      *sPtr = newPtr ;                }
    else
    {  previousPtr->nextPtr = newPtr;
      newPtr->nextPtr = currentPtr;  }
  }
  else
    printf( "%c not inserted. No memory available\n", value );
}
```


Example 9A (cont')

```
/* Delete a list element */
char deletenode( ListNodePtr *sPtr, char value )
{ ListNodePtr previousPtr , currentPtr, tempPtr;
  if ( value == ( *sPtr )->data ) {
    tempPtr = *sPtr;
    *sPtr = ( *sPtr )->nextPtr;  /* de-thread the node */
    free( tempPtr );             /* free the de-threaded node */
    return value;
  }
  else {
    previousPtr = *sPtr;
    currentPtr = ( *sPtr )->nextPtr;
    while ( currentPtr != NULL && currentPtr->data != value ) {
      previousPtr = currentPtr;        /* walk to ...  */
      currentPtr  = currentPtr->nextPtr;  /* ... next node */        }
    if ( currentPtr  != NULL ) {
      tempPtr = currentPtr ;
      previousPtr->nextPtr = currentPtr->nextPtr;
      free(  tempPtr );
      return value;               }
  }
  return \0;
}
```

Example 9A (cont')

```
/* Return 1 if the list is empty, 0 otherwise */
int isEmpty ( ListNodePtr sPtr )
{
   return sPtr == NULL;
}

/* Print the list */
void printList ( ListNodePtr currentPtr )
{
   if ( currentPtr == NULL )
      printf( "List is empty.\n\n" );
   else {
      printf( "The list is:\n" );

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


Output Result