(2-1) Data Structures & The Basics of a Linked List I

Instructor - Andrew S. O'Fallon
CptS 122 (January 17, 2024)
Washington State University

How do we Select a Data Structure? (1)

- Select a data structure as follows:
  - Analyze the problem and requirements to determine the resource constraints for the solution
  - Determine basic operations that must be supported
    - Quantify resource constraints for each operation
  - Select the data structure that best fits these requirements/constraints

- Courtesy of Will Thacker, Winthrop University

A. O'Fallon, J. Hagemeister

How do we Select a Data Structure? (2)

- Questions that must be considered:
  - Is the data inserted into the structure at the beginning or the end? Or are insertions interspersed with other operations?
  - Can data be deleted?
  - Is the data processed in some well-defined order, or is random access allowed?

- Courtesy of Will Thacker, Winthrop University

Other Considerations for Data Structures? (1)

- Each data structure has costs and benefits
- Rarely is one data structure better than another in all situations
- A data structure requires:
  - Space for each data item it stores,
  - Time to perform each basic operation,
  - Programming effort

- Courtesy of Will Thacker, Winthrop University

Other Considerations for Data Structures? (2)

- Each problem has constraints on available time and space
- Only after a careful analysis of problem characteristics can we know the best data structure for the task

- Courtesy of Will Thacker, Winthrop University

The List ADT

A. O'Fallon, J. Hagemeister

Definition of Linked List

- A finite sequence of nodes, where each node may be only accessed sequentially (through links or pointers), starting from the first node
- It is also defined as a linear collection of self-referential structures connected by pointers

A. O'Fallon, J. Hagemeister

Conventions

- An uppercase first character of a function name indicates that we are referencing the List ADT operation
- A lowercase first character of a function indicates our implementation

Struct Node

- For these examples, we'll use the following definition for `Node`:

```
typedef struct node
{
    char data;
    // self-referential
    struct node *pNext;
} Node;
```


A. O'Fallon, J. Hagemeister

Initializing a List (1)

- **InitList (L)** Procedure to initialize the list L to empty
- Our implementation:

```
void initList (Node **pList)
{
    // Recall: we must dereference a
    // pointer to retain changes
    *pList = NULL;
}
```


A. O'Fallon, J. Hagemeister

Initializing a List (2)

- The `initList()` function is elementary and is not always implemented
- We may instead initialize the pointer to the start of the list with `NULL` within `main()`

```
int main (void)
{
    Node *pList = NULL;
    …
}
```


A. O'Fallon, J. Hagemeister

Checking for Empty List (1)

- **ListIsEmpty (L) -> b:** Boolean function to return TRUE if L is empty
- Our implementation:

```
int isEmpty (Node *pList)
{
    int status = 0; // False initially

    if (pList == NULL) // The list is empty
    {
          status = 1; // True
    }

    return status;
}
```


A. O'Fallon, J. Hagemeister

Checking for Empty List (2)

- Note: we could substitute the `int` return type with an enumerated type such as Boolean

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


A. O'Fallon, J. Hagemeister

Checking for Empty List (3)

- Our implementation with `Boolean` defined:

```
Boolean isEmpty (Node *pList)
{
     Boolean status = FALSE;

     if (pList == NULL)
     {
           status = TRUE;
     }

     return status;
}
```


A. O'Fallon, J. Hagemeister

Printing Data in List (1)

- Our implementation:

```
void printListIterative (Node *pList)
{
    printf ("X -> ");
    while (pList != NULL)
    {
        printf ("%c -> ", pList -> data);
        // Get to the next item
        pList = pList -> pNext;
    }
    printf ("NULL\n");
}
```


A. O'Fallon, J. Hagemeister

Printing Data in List (2)

- Another possible implementation using `isEmpty()`:

```
void printListIterative (Node *pList)
{
     printf (“X -> “);
     while (!isEmpty (pList))
     {
            printf (“%c -> “, pList -> data);
            // Get to the next item
            pList = pList -> pNext;
     }
     printf (“NULL\n”);
}
```


A. O’Fallon, J. Hagemeister

Printing Data in List (3)

- We can determine the end of the list by searching for the `NULL` pointer
- If the list is initially empty, no problem, the `while()` loop will not execute

A. O'Fallon, J. Hagemeister

Inserting Data at Front of List

- **InsertFront (L,e):** Procedure to insert a node with information e into L as the first node in the List; in case L is empty, make a node containing e the only node in L and the current node

A. O'Fallon, J. Hagemeister

Inserting Data at Front of List w/o Error Checking (1)

- Our implementation:

```
void insertFront (Node **pList, char newData)
{
     Node *pMem = NULL;

     pMem = (Node *) malloc (sizeof (Node));
     // Initialize the dynamic memory
     pMem -> data = newData;
     pMem -> pNext = NULL;

     // Insert the new node into front of list
     pMem -> pNext = *pList;
     *pList = pMem;

}
```

Inserting Data at Front of List w/o Error Checking (2)

- Let’s define a new function which handles the dynamic allocation and initialization of a node:

```
Node * makeNode (char newData)
{
     Node *pMem = NULL;

     pMem = (Node *) malloc (sizeof (Node));
     // Initialize the dynamic memory
     pMem -> data = newData;
     pMem -> pNext = NULL;

     return pMem;
}
```


A. O’Fallon, J. Hagemeister

Inserting Data at Front of List w/o Error Checking (3)

- Now we can reorganize our code and take advantage of the new function:

```
void insertFront (Node **pList, char newData)
{
    Node *pMem = NULL;

    pMem = makeNode (newData);

    // Insert the new node into front of list
    pMem -> pNext = *pList;
    *pList = pMem;

}
```


A. O'Fallon, J. Hagemeister

Inserting Data at Front of List w/ Error Checking (1)

- Let's modify our code so that we can check for dynamic memory allocation errors
- We'll start with `makeNode()`:

```
Node * makeNode (char newData)
{
     Node *pMem = NULL;

     pMem = (Node *) malloc (sizeof (Node));
     if (pMem != NULL)
     {
          // Initialize the dynamic memory
          pMem -> data = newData;
          pMem -> pNext = NULL;
     }
     // Otherwise no memory is available; could use else, but
     // it's not necessary

     return pMem;
}
```


A. O'Fallon, J. Hagemeister

Inserting Data at Front of List w/ Error Checking (2)

- Now let's add some error checking to `insertFront ():`

```
void insertFront (Node **pList, char newData)
{
        Node *pMem = NULL;

        pMem = makeNode (newData);

        if (pMem != NULL) // Memory was available
        {
                // Insert the new node into front of list
                pMem -> pNext = *pList;
                *pList = pMem;
        }
        else // Can't allocate anymore dynamic memory
        {
            printf ("WARNING: No memory is available for data insertion!\n")
        }

}
```


A. O'Fallon, J. Hagemeister

Closing Thoughts

- Can you build a driver program to test these functions?
- Is it possible to return a `Boolean` for `insertFront()` to indicate a memory allocation error, where TRUE means error and FALSE means no error?
- `insertFront()` will be seen again with a Stack data structure…

Next Lecture…

- Continue our discussion and implementation of linked lists

A. O'Fallon, J. Hagemeister

References

- P.J. Deitel & H.M. Deitel, *C: How to Program* (8th ed.), Prentice Hall, 2017
- J.R. Hanly & E.B. Koffman, *Problem Solving and Program Design in C (7th Ed.)*, Addison-Wesley, 2013

A. O'Fallon, J. Hagemeister

Collaborators

- Jack Hagemeister