Chapter 12: Dynamic Data Structure

Introduction

* The sizes fo dynamic data structures can grow abd shrink at execution time.
* Linked lists: collections of data items "lined up in a row"; insertions and deletions are made anywhere in a linked list.
* Stacks are important in compilers and operating systems; insertions and deletions are made only at one end of stack, which is top
* Queues represent waiting lines; insertions are made at the back (also referred to as the tail) of a queue, and deletions are made from the front (also referred as the head) of the queue.
* Binary tree facilitate high-speed searching and sorting of data, efficient elimination of duplicate data items, representing file system directories, and compiling expressions into machine language.

Self-Referential Structures

* it contains a pointer member that points to a structure of the same structure type.
* E.g.

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

* member "nextPtr" points to a structure of type struct node - a structure of the same type as the one being declared
* "nextPtr" is referred to as a link, i.e., "nextPtr" can be used to "tie" a structure of type "struct node" to another structure of the same type.
* this is the basic form of lists, queues, stacks and trees
* the relationship can be shown by

```
 [15|*]-->[10|\]
```

* a slash - representing a "NULL" pointer - is placed in the link member of the last self-referential structure to indicate that the link does not point to another structure.

Dynamic Memory Allocation

* the ability for a program to obtain more memory space at execution time to hold new nodes, and to release space no longer needed

* Function "malloc" takes as an argument the number of bytes to be allocated, and return a pointer of type "void *" (pointer to void) to the allocated memory
* e.g.

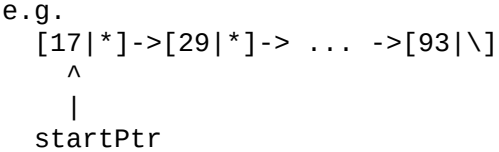
```
    newPtr = malloc(sizeof(struct node));
```

* the "sizeof(struct node)" is evaluted to determine the size in bytes of a structure of type "struct node", allocates a new area of memory for "sizeof(struct node)" byte, and stores a pointer to the allocated memory in variable "newPtr". If no memory is available, "malloc" returns a "NULL" pointer

* Function "free" deallocates memory, i.e., the memory is returned to the system so that the memory can be reallocated in the future. To free memory dynamically allocated by the preceding "malloc" call, use

the statement,
* e.g.
```
      free(newPtr);
```

Linked Lists
---
* a linked list is a linear collection of self-referential structures, called nodes, connected by pointer links
* a linked list is accessed via a pointer to the first node of the list
* subsequent nodes are accessed via the link pointer member stored in each node
* the link pointer in the last node of a list is set to "NULL" to mark the end of the list
* data are stored dynamically; each node is created as necessary.
* linked list is appropriate when the number of data elements to be represented in the data structure at once is unpredictable; the length of a list can increase ot decrease as necessary
* the size of array cannot be altered, because array memory is allocated at compile time
* arrarys can become full; linked lists become full when the system has insufficient memory to satisfy dynamic storage allocation requests
* linked lists can be maintained in sorted order by inserting each new element at the proper point in the list
* e.g.



Main Program
---
* The main program manipulates a list of charachers is shown in the following.
* The program is used to insert a character in the list in alphabetical order and delete a character from the list.

* e.g.

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

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

        typedef struct listNode LISTNODE;
        typedef LISTNODE *LISTNODEPTR;

        void insert(LISTNODEPTR *, char);
        char delete(LISTNODEPTR *, char);
        int isEmpty(LISTNODEPTR);
        void printList(LISTNODEPTR);
        void instructions(void);

        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);
            insert(&startPtr, item);
            printList(startPtr);
            break;
         case 2:
            if (!isEmpty(startPtr)) {
               printf("Enter character to be deleted: ");
               scanf("\n%c", &item);

               if (delete(&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);
   }

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

isEmpty

```
-------
int isEmpty(LISTNODEPTR sPtr)
{
   return sPtr == NULL;
```

```
        }
```

* Function isEmpty determines if the list is empty. If the list is empty, 1 is return; otherwise, 0 is returned

printList

```
        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");
           }
        }
```

* Function printList prints the list.
* it receives a pointer to the start of the list as an argument, and refers to the pointer as currentPtr.
* The function first determines if the list is empty.
* Otherwise, it prints the data in the list. While "currentPtr" is not "NULL", "currentPtr->data" is printed by the function, and "currentPtr->nextPtr" is assigned to "currentPtr".
* Note that if the link in the last of the list is not NULL, the printing algorithm will try to print past the end of the list, and an error will occur.

Insert

```
        void insert(LISTNODEPTR *sPtr, char value)
        {
           LISTNODEPTR newPtr, previousPtr, currentPtr;

           newPtr = malloc(sizeof(LISTNODE));

           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);
      }
```

* Functions insert (and delete) receives the address of the list and a character to be inserted (or deleted). The address of the list is necessary when a value is to be inserted at the start of the list.
* Providing the address of the list enables the list (i.e. the pointer to the first node of the list) to be modified via a call by reference.

* The steps are as follows:
1) Create a node by calling malloc, assigning to newPtr the address of the allocated memory, assigning the character to be inserted to newPtr->data, and assigning NULL to newPtr->nextPtr

2) Initialize previousPtr to NULL, and currentPtr to *sPtr (the pointer to the start of the list). Pointers previousPtr and currentPtr are used to store the locations of the node preceding the insertion point and the node after the insertion point

3) While currentPtr is not NULL and the value to be inserted is greater than currentPtr->data, assign currentPtr to previousPtr and advance currentPtr to the next node in the list. This locates the insertion point for the value in the list.

4) If previousPtr is NULL, the new node is inserted as the first node in the list. Assign *sPtr to newPtr->nextPtr (the new node link points to the former first node), and assign newPtr to *sPtr (*sPtr points to the new node). If previousPtr is not NULL, the new node is inserted in place. Assign newPtr to previousPtr->nextPtr (the previos node points to the new node), and assign currentPtr to newPtr->nextPtr (the new node link points to the current node).

Delete

```
    char delete(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';
}
```

* Function delete receives the address of the pointer to the start of the list and a character to be deleted. The steps for deleting a character from the list are as follows:

1) If the character to be deleted matches the character in the first node of the list, assign *sPtr to tempPtr (tempPtr will be used to free the unneeded memory), assign (*sPtr)->nextPtr to *sPtr (*sPtr now points to the second node in the list), free the memory pointed to by tempPtr, and return the character that was deleted.

2) Otherwise, initialize previousPtr with *sPtr and initialize currentPtr with (*sPtr)->nextPtr

3) While currentPtr is not NULL and the value to be deleted is not equal to currentPtr->data, assign currentPtr to previousPtr, and assign currentPtr->nextPtr to currentPtr. This locates the character to be deleted if it is contained in the list

4) If currentPtr is not NULL, assign currentPtr to tempPtr, assign currentPtr->nextPtr to previousPtr->nextPtr, free the node pointed to by tempPtr, and return the character that was deleted from the list. If currentPtr is NULL, return the NULL character ('\0') to signify that the character to be deleted was not found in the list

Dummy Head Version

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

struct listNode
{
    char data;
    struct listNode *nextPtr;
};

typedef struct listNode LISTNODE;
typedef LISTNODE *LISTNODEPTR;

void insert(LISTNODEPTR, char);
char delete(LISTNODEPTR, char);
int isEmpty(LISTNODEPTR);
void printList(LISTNODEPTR);
void instructions(void);
```

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

    dump = malloc(sizeof(LISTNODE));
    dump->nextPtr = NULL;
    startPtr = dump;

    instructions();
    printf("? ");
    scanf("%d", &choice);

    while (choice != 3)
    {
        switch (choice)
        {
            case 1:
                printf("Enter a character: ");
                scanf("\n%c", &item);
                insert(startPtr, item);
                printList(startPtr);
                break;
            case 2:
                if (!isEmpty(startPtr))
                {
                    printf("Enter character to be deleted: ");
                    scanf("\n%c", &item);

                    if (delete(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);
    }

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

```
void instructions()
{
    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");
}

void insert(LISTNODEPTR sPtr, char value)
{
    LISTNODEPTR newPtr, previousPtr, currentPtr;

    newPtr = malloc(sizeof(LISTNODE));

    if (newPtr != NULL)
    {
        newPtr->data = value;
        newPtr->nextPtr = NULL;

        previousPtr = sPtr;
        currentPtr = sPtr->nextPtr;

        while (currentPtr != NULL && value > currentPtr->data)
        {
            previousPtr = currentPtr;
            currentPtr = currentPtr->nextPtr;
        }

        previousPtr->nextPtr = newPtr;
        newPtr->nextPtr = currentPtr;
    }
    else
    {
        printf("%c not inserted. No memory available.\n",value);
    }
}

char delete(LISTNODEPTR sPtr, char value)
{
    LISTNODEPTR previousPtr, currentPtr, tempPtr;

    previousPtr = sPtr;
    currentPtr = sPtr->nextPtr;

    while (currentPtr != NULL && currentPtr->data != value)
    {
        previousPtr = currentPtr;
        currentPtr = currentPtr->nextPtr;
    }

    if (currentPtr != NULL)
    {
        tempPtr = currentPtr;
        previousPtr->nextPtr = currentPtr->nextPtr;
        free(tempPtr);
        return value;
    }

    return '\0';
```

```
}

int isEmpty(LISTNODEPTR sPtr)
{
    return (sPtr->nextPtr == NULL);
}

void printList(LISTNODEPTR currentPtr)
{
    currentPtr = currentPtr->nextPtr;

    if (currentPtr == NULL)
    {
        printf("List is empty.\n\n");
    }
    else
    {
        printf("This list is: ");

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

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

Execrise
========

1. Modify the delete procedure to delete all nodes of the specified character.

2. Write a recursive and iterative procedures to search a linked list for a particular data item.