Lesson 2:
Linked Lists

2.1

2.1 Linked Lists

Introduction

- One of most fundamental data structures
- Elements linked together in specific order
- Advantages vs. arrays
    - O(1) inserts (arrays have O(n))
    - O(1) deletes (arrays have O(n))
    - Dynamically sized (arrays require # elems to be known before array created)
    - Do not require contiguous memory
- Disadvantages vs. arrays
    - O(n) random access (arrays have O(1))
    - Uses additional memory linking each element to next

Definitions

- Head: Pointer to first element in list
- Tail: Pointer to last element in list
- Singly-linked lists
    - Each element links to next element
    - Can be traversed forward only
- Doubly-linked lists
    - Each element links to next *and previous* element
    - Can be traversed forward *and backward*
- Circular lists
    - List that wraps around on itself
    - Has no beginning or end
    - Can be implemented using a singly-linked or doubly-linked list

Applications

- Well-suited for solving these kinds of problems:
    - Order of elements matters
    - Random-access not required
    - Number of elements unknown beforehand
- Examples:
    - Mailing lists
        - Size of mailing list may not be known beforehand

        - Mailer builds list of email addresses before sending message
    - Scrolled lists (in GUI)
        - Limited set of items in scrolled list shown at any time
        - One way to represent items available is to store them in list
    - Polynomials
        - Each element in list stores one term
        - e.g. 3x^2 + 2x + 1 would be represented by 3 nodes: 3 → 2 → 1
    - Memory management
        - Heap management software can track blocks of allocated memory in list
        - Linked list good for this application since elements inserted & deleted frequently
    - Linked allocation of files
        - Large files spread across multiple blocks
        - Each block links to next block
    - Other data structures
        - Other data structures use lists internally to hold data
        - e.g. stacks, queues, sets, hash tables, graphs (we'll see this later)

2.2 Singly-Linked Lists

Overview

- Each element links to next element via a pointer
- Head pointer points to first element in list
- Tail pointer points to last element in list
- Last element points to NULL
- Can only be traversed from head to tail
- If need to maintain position in list, we must maintain pointer to element
- Conceptually elements ordered
- Physically elements stored non-contiguously in memory
- Fast
    - Inserts
    - Deletes
- Slow
    - Random-access

Visualization – Layout in memory

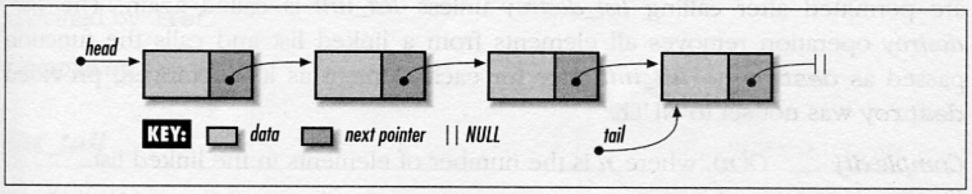


Figure 5-1. Elements linked together to form a linked list

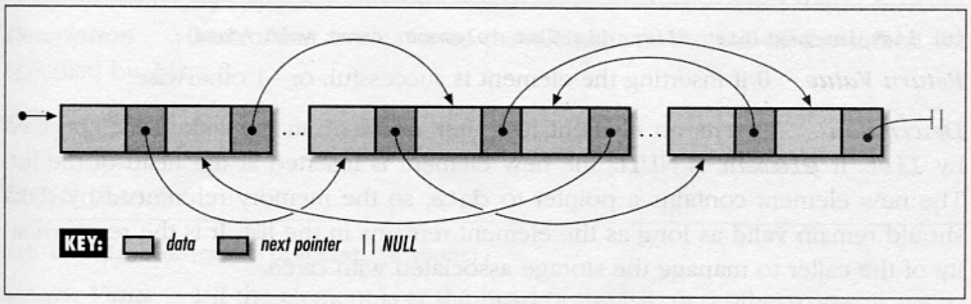


Figure 5-2. Elements of a linked list linked but scattered about an address space

Visualization – Inserting

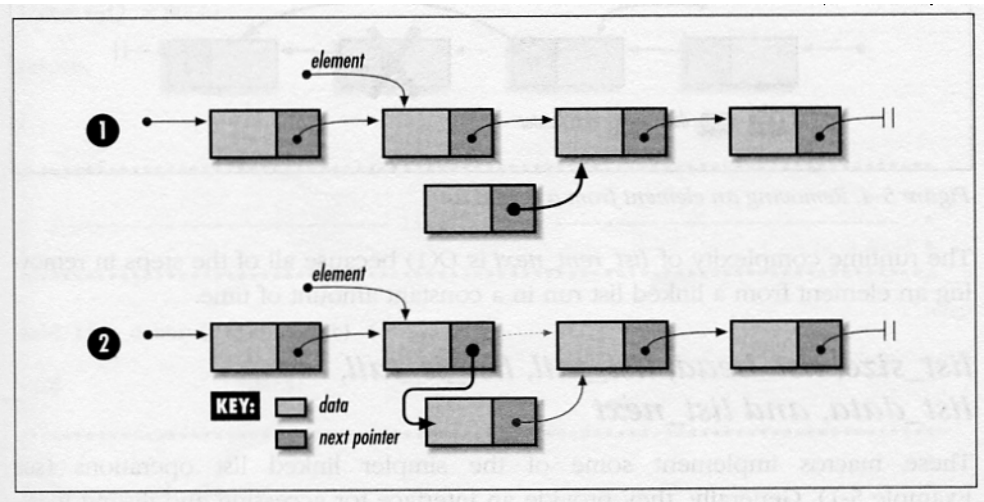
- 

Figure 5-3. Inserting an element into a linked list

Visualization – Removing

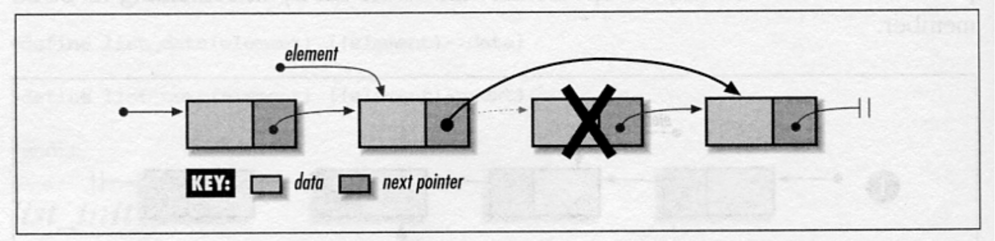
- 

Figure 5-4. Removing an element from a linked list

2.3 Singly-Linked List Implementation

Interface

list.h

```c
/*
 * list.h
 */
#ifndef LIST_H
#define LIST_H

#include <stdlib.h>

/*
 * Singly-linked list element
 */
typedef struct ListElmt_
{
    void                *data;
    struct ListElmt_    *next;
} ListElmt;

/*
 * Singly-linked list
 */
typedef struct List_
{
    int                 size;

    int                 (*match)(const void *key1, const void *key2);
    void                (*destroy)(void *data);

    ListElmt            *head;
    ListElmt            *tail;
} List;

/*
 * Public interface
 */
void list_init(List *list, void (*destroy)(void *data));

void list_destroy(List *list);

int list_ins_next(List *list, ListElmt *element, const void *data);

int list_rem_next(List *list, ListElmt *element, void **data);

#define list_size(list) ((list)->size)

#define list_head(list) ((list)->head)

#define list_tail(list) ((list)->tail)

#define list_is_head(list, element) ((element) == (list)->head ? 1 : 0)

#define list_is_tail(element) ((element)->next == NULL ? 1 : 0)

#define list_data(element) ((element)->data)

#define list_next(element) ((element)->next)

#endif
```

- ListElmt
    - Represents an element in a singly-linked list
    - data
        - points to data stored in element
    - next
        - points to next element in list
        - points to NULL if last element in list

- List
    - Represents a singly-linked list
    - size
        - number of elements in the list
    - match
        - pointer to function used to compare elements in the list
        - used by some algorithms (will see this used later in class)
    - destroy

        - pointer to function called on each data in list when list is destroyed
    - head
        - pointer to first element in list
        - points to NULL if list is empty
    - tail
        - pointer to last element in list
        - points to NULL if list is empty

- **[See pages 53-56 in book]**

Implementation

list.c

```c
/*
 * list.c
 */
#include <stdlib.h>
#include <string.h>

#include "list.h"

void list_init(List *list, void (*destroy)(void *data))
{
    /* Initialize the list */
    list->size = 0;
    list->destroy = destroy;
    list->head = NULL;
    list->tail = NULL;
}

void list_destroy(List *list)
{
    void *data;

    /* Remove each element */
    while (list_size(list) > 0) {
        if (list_rem_next(list, NULL, (void **)&data) == 0 && list->destroy !=
                NULL) {
            /* Call a user-defined function to free dynamically allocated
```

```
               data. */
            list->destroy(data);
        }
    }

    /* No operations are allowed now, but clear the structure as a
       precaution. */
    memset(list, 0, sizeof(List));
}

int list_ins_next(List *list, ListElmt *element, const void *data)
{
    ListElmt *new_element;

    /* Allocate storage for the element. */
    if ((new_element = (ListElmt *)malloc(sizeof(ListElmt))) == NULL)
        return -1;

    /* Insert the element into the list. */
    new_element->data = (void *)data;
    if (element == NULL) {
        /* Handle insertion at the head of the list. */
        if (list_size(list) == 0)
            list->tail = new_element;
        new_element->next = list->head;
        list->head = new_element;
    }
    else {
        /* Handle insertion somewhere other than at the head. */
        if (element->next == NULL)
            list->tail = new_element;
        new_element->next = element->next;
        element->next = new_element;
    }

    /*  Adjust the size of the list to account for the inserted element. */
    list->size++;

    return 0;
}

int list_rem_next(List *list, ListElmt *element, void **data)
```

```c
{
    ListElmt *old_element;

    /* Do not allow removal from an empty list. */
    if (list_size(list) == 0)
        return -1;

    /* Remove the element from the list. */
    if (element == NULL) {
        /* Handle removal from the head of the list. */
        *data = list->head->data;
        old_element = list->head;
        list->head = list->head->next;

        if (list_size(list) == 1)
            list->tail = NULL;
    }
    else {
        /* Handle removal from somewhere other than the head. */
        if (element->next == NULL)
            return -1;

        *data = element->next->data;
        old_element = element->next;
        element->next = element->next->next;

        if (element->next == NULL)
            list->tail = element;
    }

    /* Free the storage allocated by the abstract data type. */
    free(old_element);

    /* Adjust the size of the list to account for the removed element. */
    list->size--;

    return 0;
}
```

2.4 Singly-Linked List Example #1

Inserting numbers into a singly-linked list

```
/*
 * File:  insert-numbers-into-list.c
 *
 * Program demonstrating inserting random numbers at tail of singly-linked
 * list.
 */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#include "list.h"

int main()
{
    List listOfInts;
    ListElmt *pElmt;
    int *pInt;
    int i;

    /* Seed the random number generator */
    srand(time(NULL));

    /* Initialize list */
    list_init(&listOfInts, free);

    /* Output number of elements in list */
    printf("Before inserts - list size: %d\n", list_size(&listOfInts));

    /* Insert some data at tail of list */
    for (i = 0; i < 5; ++i) {
        /* Dynamically allocate an int */
        pInt = (int *)malloc(sizeof(int));
        *pInt = random() % 100;

        /* Insert the int at tail of list */
        printf("Inserting %d into list\n", *pInt);
        list_ins_next(&listOfInts, list_tail(&listOfInts), pInt);
    }
```

```c

    /* Output number of elements in list */
    printf("After inserts - list size: %d\n", list_size(&listOfInts));

    /* Output the data in the list */
    printf("Elements:");
    pElmt = list_head(&listOfInts);
    while (pElmt != NULL) {
        pInt = (int *)list_data(pElmt);
        printf(" %d", *pInt);
        pElmt = list_next(pElmt);
    }
    printf("\n");

    /* Destroy the list (this automatically calls destroyInt on each data) */
    list_destroy(&listOfInts);

    /* Output number of elements in list */
    printf("After destroy - list size: %d\n", list_size(&listOfInts));

    return EXIT_SUCCESS;
}

/* Program output:
        Before inserts - list size: 0
        Inserting 83 into list
        Inserting 86 into list
        Inserting 77 into list
        Inserting 15 into list
        Inserting 93 into list
        After inserts - list size: 5
        Elements: 83 86 77 15 93
        After destroy - list size: 0
 */
```

2.5 Singly-Linked List Example #2

Merging two sorted singly-linked lists

```c
/*
 * File:  merge-two-sorted-lists.c
 *
 * Demonstrate merging two ordered singly-linked lists into
 * one ordered singly-linked list.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "list.h"

void loadDataIntoList(List *pList, const char *listData[], int listDataLength);
void mergeSortedLists(const List *pList1, const List *pList2, List *pDestList);
void outputList(List *pList);

int main()
{
   /* Variables */
   const char *list1Data[] = {
      "aardvark",
      "badger",
      "horse",
      "zebra"
   };
   const char *list2Data[] = {
      "armadillo",
      "baboon",
      "cat",
      "kangaroo"
   };
   List list1;
   List list2;
   List mergedList;

   /* Initialize the lists */
   list_init(&list1, NULL);
   list_init(&list2, NULL);
   list_init(&mergedList, NULL);

   /* Load sorted data into lists */
   loadDataIntoList(&list1, list1Data, sizeof(list1Data) / sizeof(*list1Data));
   loadDataIntoList(&list2, list2Data, sizeof(list2Data) / sizeof(*list2Data));

   /* Merge sorted list into destination list */
   mergeSortedLists(&list1, &list2, &mergedList);

   /* Output contents of all three lists */
   outputList(&list1);
   outputList(&list2);
   outputList(&mergedList);

   /* Destroy the lists */
   list_destroy(&list1);
   list_destroy(&list2);
   list_destroy(&mergedList);

   return EXIT_SUCCESS;
}

void loadDataIntoList(List *pList, const char *listData[], int listDataLength)
{
   int i;

   for (i = 0; i < listDataLength; ++i)
   {
      list_ins_next(pList, list_tail(pList), listData[i]);
   }
}

void mergeSortedLists(const List *pList1, const List *pList2, List *pDestList)
{
   ListElmt *pList1Elmt;
   ListElmt *pList2Elmt;
   const char *pList1Data;
   const char *pList2Data;
   const char *pDataToInsert;

   /* Get head element from both source lists */
   pList1Elmt = list_head(pList1);
   pList2Elmt = list_head(pList2);

   /* Add all elements from both lists to the destination list while
      maintaining sorted order */
   while (pList1Elmt != NULL || pList2Elmt != NULL)
   {
      /* Determine which data should be added next */
      if (pList1Elmt == NULL)
      {
         /* Everything from list 1 has been added, select next from list 2 */
         pDataToInsert = (const char *)list_data(pList2Elmt);
         pList2Elmt = list_next(pList2Elmt);
      }
      else if (pList2Elmt == NULL)
      {
         /* Everything from list 2 has been added, select next from list 1 */
         pDataToInsert = (const char *)list_data(pList1Elmt);
         pList1Elmt = list_next(pList1Elmt);
      }
      else
      {
         /* List 1 and 2 contain more elements */
         pList1Data = (const char *)list_data(pList1Elmt);
         pList2Data = (const char *)list_data(pList2Elmt);

         /* Determine whether element from list 1 or 2 should be added next */
         if (strcmp(pList1Data, pList2Data) < 0)
         {
            /* List 1 contains smaller element, select next from list 1 */
            pDataToInsert = (const char *)list_data(pList1Elmt);
            pList1Elmt = list_next(pList1Elmt);
         }
         else
         {
            /* List 2 contains smaller element, select next from list 2 */
            pDataToInsert = (const char *)list_data(pList2Elmt);
            pList2Elmt = list_next(pList2Elmt);
         }
      }

      /* Add the data to the end of the destination list */
      list_ins_next(pDestList, list_tail(pDestList), pDataToInsert);
```

```c
   }
}

void outputList(List *pList)
{
   ListElmt *pListElmt;

   printf("List\n");
   printf("----\n");

   /* Get head element of list */
   pListElmt = list_head(pList);

   /* Output each element */
   while (pListElmt != NULL)
   {
      printf("%s\n", (const char *)list_data(pListElmt));
      pListElmt = list_next(pListElmt);
   }

   printf("\n");
}

/*
 * Program output:
        List
        ----
        aardvark
        badger
        horse
        zebra

        List
        ----
        armadillo
        baboon
        cat
        kangaroo

        List
        ----
        aardvark
        armadillo
        baboon
        badger
        cat
        horse
        kangaroo
        zebra
*/
```

2.6 Doubly-Linked Lists

Overview

- Same as singly-linked list except for following…
- Each element links to next *and previous* elements via pointers
- Last element's *next* pointer points to NULL
- First element's *previous* pointer points to NULL
- Can be traversed from head to tail *or from tail to head*

Visualization – Inserting

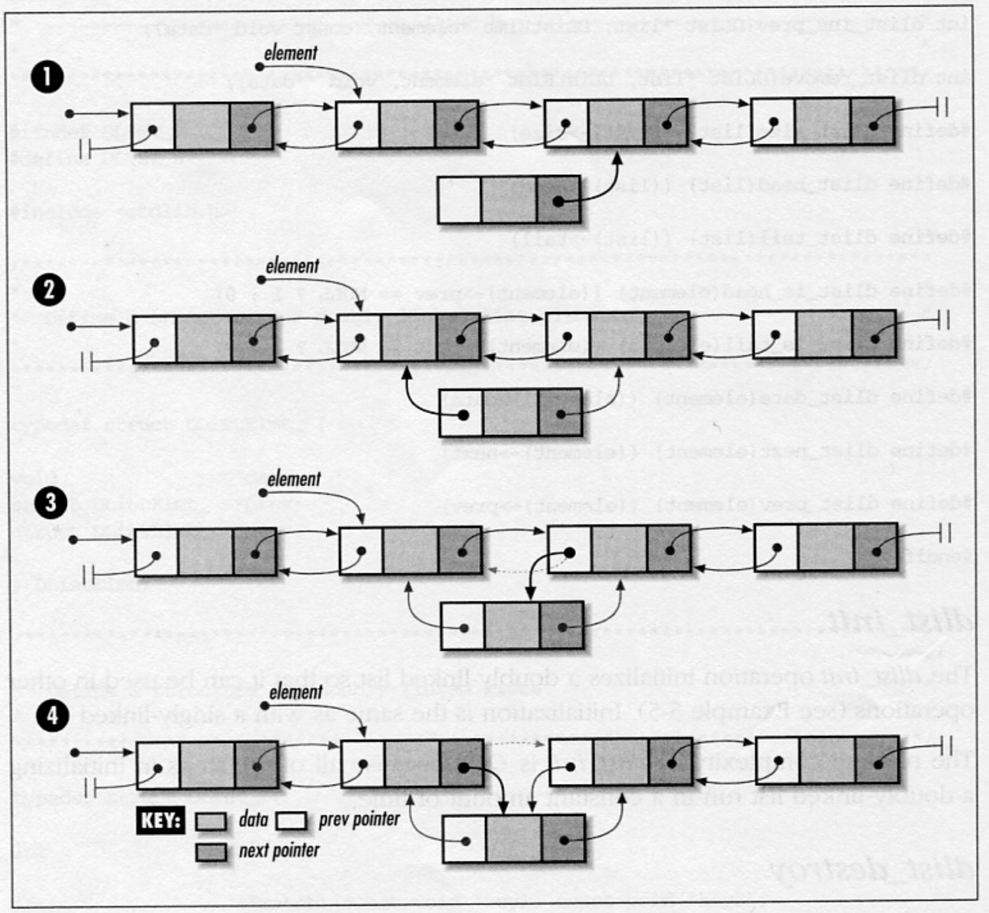


Figure 5-6. Inserting an element into a doubly-linked list with dlist_ins_next

2.7 Doubly-Linked List Implementation

Interface

dlist.h

```c
/*
 * dlist.h
 */
#ifndef DLIST_H
#define DLIST_H

#include <stdlib.h>

/*
 * Doubly-linked list element
 */
typedef struct DListElmt_
{
    void                *data;
    struct DListElmt_   *prev;
    struct DListElmt_   *next;
} DListElmt;

/*
 * Doubly-linked list
 */
typedef struct DList_
{
    int                 size;

    int                 (*match)(const void *key1, const void *key2);
    void                (*destroy)(void *data);

    DListElmt           *head;
    DListElmt           *tail;
} DList;

/*
 * Public interface
 */
void dlist_init(DList *list, void (*destroy)(void *data));

void dlist_destroy(DList *list);

int dlist_ins_next(DList *list, DListElmt *element, const void *data);

int dlist_ins_prev(DList *list, DListElmt *element, const void *data);

int dlist_remove(DList *list, DListElmt *element, void **data);

#define dlist_size(list) ((list)->size)

#define dlist_head(list) ((list)->head)

#define dlist_tail(list) ((list)->tail)

#define dlist_is_head(element) ((element)->prev == NULL ? 1 : 0)

#define dlist_is_tail(element) ((element)->next == NULL ? 1 : 0)

#define dlist_data(element) ((element)->data)

#define dlist_next(element) ((element)->next)

#define dlist_prev(element) ((element)->prev)

#endif
```

- DListElmt
    - Represents an element in a doubly-linked list
    - data
        - points to data stored in element
    - next
        - points to next element in list
        - points to NULL if last element in list
    - prev
        - points to previous element in list
        - points to NULL if first element in list

- DList

    - Represents a doubly-linked list
    - size
        - number of elements in the list
    - match
        - pointer to function used to compare elements in the list
        - used by some algorithms (will see this used later in class)
    - destroy
        - pointer to function called on each data in list when list is destroyed
    - head
        - pointer to first element in list
        - points to NULL if list is empty
    - tail
        - pointer to last element in list
        - points to NULL if list is empty

- **[See pages 68-71 in book]**

Implementation

dlist.c

```
/*
 * dlist.c
 */
#include <stdlib.h>
#include <string.h>

#include "dlist.h"

void dlist_init(DList *list, void (*destroy)(void *data))
{
    /* Initialize the list. */
    list->size = 0;
    list->destroy = destroy;
    list->head = NULL;
    list->tail = NULL;
}

void dlist_destroy(DList *list)
```

```
{
    void *data;

    /* Remove each element. */
    while (dlist_size(list) > 0) {
        if (dlist_remove(list, dlist_tail(list), (void **)&data) == 0 &&
                list->destroy != NULL) {
            /* Call a user-defined function to free dynamically allocated
               data. */
            list->destroy(data);
        }
    }

    /* No operations are allowed now, but clear the structure as a
       precaution. */
    memset(list, 0, sizeof(DList));
}

int dlist_ins_next(DList *list, DListElmt *element, const void *data)
{
    DListElmt *new_element;

    /* Do not allow a NULL element unless the list is empty. */
    if (element == NULL && dlist_size(list) != 0)
        return -1;

    /* Allocate storage for the element. */
    if ((new_element = (DListElmt *)malloc(sizeof(DListElmt))) == NULL)
        return -1;

    /* Insert the new element into the list. */
    new_element->data = (void *)data;

    if (dlist_size(list) == 0) {
        /* Handle insertion when the list is empty. */
        list->head = new_element;
        list->head->prev = NULL;
        list->head->next = NULL;
        list->tail = new_element;
    }
    else {
        /* Handle insertion when the list is not empty. */
        new_element->next = element->next;
        new_element->prev = element;

        if (element->next == NULL)
            list->tail = new_element;
        else
            element->next->prev = new_element;

        element->next = new_element;
    }

    /* Adjust the size of the list to account for the inserted element. */
    list->size++;

    return 0;
}

int dlist_ins_prev(DList *list, DListElmt *element, const void *data)
{
    DListElmt *new_element;

    /* Do not allow a NULL element unless the list is empty. */
    if (element == NULL && dlist_size(list) != 0)
        return -1;

    /* Allocate storage to be managed by the abstract data type. */
    if ((new_element = (DListElmt *)malloc(sizeof(DListElmt))) == NULL)
        return -1;

    /* Insert the new element into the list. */
    new_element->data = (void *)data;

    if (dlist_size(list) == 0) {
        /* Handle insertion when the list is empty. */
        list->head = new_element;
        list->head->prev = NULL;
        list->head->next = NULL;
        list->tail = new_element;
    }
    else {
        /* Handle insertion when the list is not empty. */
        new_element->next = element;
        new_element->prev = element->prev;

        if (element->prev == NULL)
            list->head = new_element;
        else
            element->prev->next = new_element;

        element->prev = new_element;
    }

    /* Adjust the size of the list to account for the new element. */
    list->size++;

    return 0;
}

int dlist_remove(DList *list, DListElmt *element, void **data)
{
    /* Do not allow a NULL element or removal from an empty list. */
    if (element == NULL || dlist_size(list) == 0)
        return -1;

    /* Remove the element from the list. */
    *data = element->data;
    if (element == list->head) {
        /* Handle removal from the head of the list. */
        list->head = element->next;

        if (list->head == NULL)
            list->tail = NULL;
        else
            element->next->prev = NULL;
    }
    else {
        /* Handle removal from other than the head of the list. */
        element->prev->next = element->next;

        if (element->next == NULL)
            list->tail = element->prev;
        else
            element->next->prev = element->prev;
    }

    /* Free the storage allocated by the abstract data type. */
    free(element);

    /* Adjust the size of the list to account for the removed element. */
    list->size--;

    return 0;
}
```

2.8 Circular Lists

Overview

- Same as singly-linked list except for following…
- Last element points to the head of the list
- There is no end of the list, so no need for Tail
- Can also be implemented as doubly-linked list in which case first element's previous pointer points to last element in list

Visualization

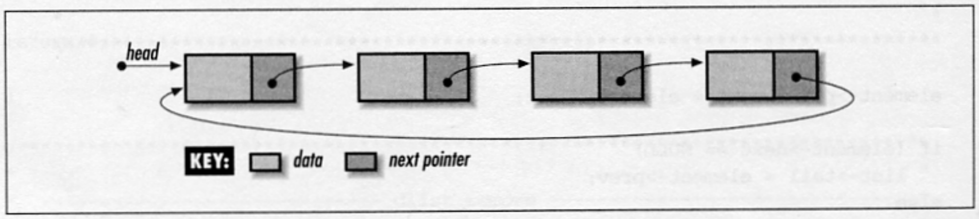
- 

Figure 5-7. Elements linked together to form a circular list

2.9 Circular List Implementation

Interface

clist.h

```
/*
 * clist.h
 */
#ifndef CLIST_H
#define CLIST_H

#include <stdlib.h>

/*
 * Circular list element
 */
typedef struct CListElmt_
{
    void                *data;
    struct CListElmt_   *next;
} CListElmt;

/*
 * Circular list
 */
typedef struct CList_
{
    int                 size;

    int                 (*match)(const void *key1, const void *key2);
    void                (*destroy)(void *data);

    CListElmt           *head;
} CList;

/*
 * Public interface
 */
void clist_init(CList *list, void (*destroy)(void *data));

void clist_destroy(CList *list);
```

```c

int clist_ins_next(CList *list, CListElmt *element, const void *data);

int clist_rem_next(CList *list, CListElmt *element, void **data);

#define clist_size(list) ((list)->size)

#define clist_head(list) ((list)->head)

#define clist_data(element) ((element)->data)

#define clist_next(element) ((element)->next)

#endif
```

- CListElmt
    - Represents an element in a circular linked list
    - data
        - points to data stored in element
    - next
        - points to next element in list
        - points to first element if last element in list

- CList
    - Represents a circular linked list
    - size
        - number of elements in the list
    - match
        - pointer to function used to compare elements in the list
        - used by some algorithms (will see this used later in class)
    - destroy
        - pointer to function called on each data in list when list is destroyed
    - head
        - pointer to first element in list
        - points to NULL if list is empty

Implementation

clist.c

```c
/*
 * clist.c
 */
#include <stdlib.h>
#include <string.h>

#include "clist.h"

void clist_init(CList *list, void (*destroy)(void *data))
{
    /* Initialize the list. */
    list->size = 0;
    list->destroy = destroy;
    list->head = NULL;
}

void clist_destroy(CList *list)
{
    void *data;

    /* Remove each element. */
    while (clist_size(list) > 0) {
        if (clist_rem_next(list, list->head, (void **)&data) == 0
                && list->destroy != NULL) {
            /* Call a user-defined function to free dynamically allocated
               data. */
            list->destroy(data);
        }
    }

    /* No operations are allowed now, but clear the structure as a
       precaution. */
    memset(list, 0, sizeof(CList));
}

int clist_ins_next(CList *list, CListElmt *element, const void *data)
{
    CListElmt *new_element;

    /* Allocate storage for the element. */
    if ((new_element = (CListElmt *)malloc(sizeof(CListElmt))) == NULL)
        return -1;

    /* Insert the element into the list. */
    new_element->data = (void *)data;

    if (clist_size(list) == 0) {
        /* Handle insertion when the list is empty. */
        new_element->next = new_element;
        list->head = new_element;
    }
    else {
        /* Handle insertion when the list is not empty. */
        new_element->next = element->next;
        element->next = new_element;
    }

    /* Adjust the size of the list to account for the inserted element. */
    list->size++;

    return 0;
}

int clist_rem_next(CList *list, CListElmt *element, void **data)
{
    CListElmt *old_element;

    /* Do not allow removal from an empty list. */
    if (clist_size(list) == 0)
        return -1;

    /* Remove the element from the list. */
    *data = element->next->data;

    if (element->next == element) {
        /* Handle removing the last element. */
        old_element = element->next;
        list->head = NULL;
   }
    else {
        /* Handle removing other than the last element. */
        old_element = element->next;
        element->next = element->next->next;
    }

    /* Free the storage allocated by the abstract data type. */
    free(old_element);

    /* Adjust the size of the list to account for the removed element. */
    list->size--;

    return 0;
}
```