

# Lesson 2:

# Linked Lists

## 2.1

## 2.1 Linked Lists

### 1 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

### 13 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

### 26 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

- 1                   ▪ Mailer builds list of email addresses before sending message
- 2           ○ Scrolled lists (in GUI)
- 3                   ▪ Limited set of items in scrolled list shown at any time
- 4                   ▪ One way to represent items available is to store them in list
- 5           ○ Polynomials
- 6                   ▪ Each element in list stores one term
- 7                   ▪ e.g.  $3x^2 + 2x + 1$  would be represented by 3 nodes:  $3 \rightarrow 2 \rightarrow$
- 8                   1
- 9           ○ Memory management
- 10                   ▪ Heap management software can track blocks of allocated
- 11                   memory in list
- 12                   ▪ Linked list good for this application since elements inserted &
- 13                   deleted frequently
- 14           ○ Linked allocation of files
- 15                   ▪ Large files spread across multiple blocks
- 16                   ▪ Each block links to next block
- 17           ○ Other data structures
- 18                   ▪ Other data structures use lists internally to hold data
- 19                   ▪ e.g. stacks, queues, sets, hash tables, graphs (we'll see this
- 20                   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

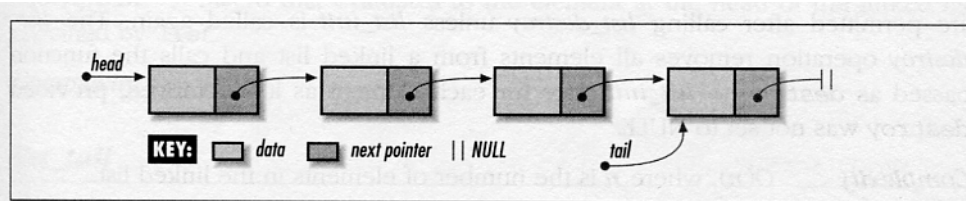


Figure 5-1. Elements linked together to form a linked list

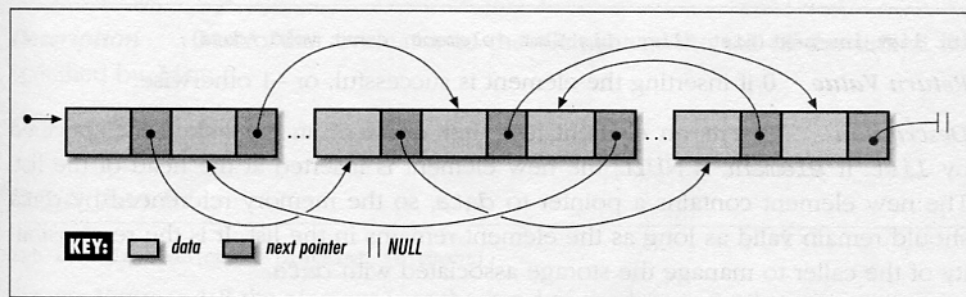


Figure 5-2. Elements of a linked list linked but scattered about an address space

## 1 Visualization – Inserting

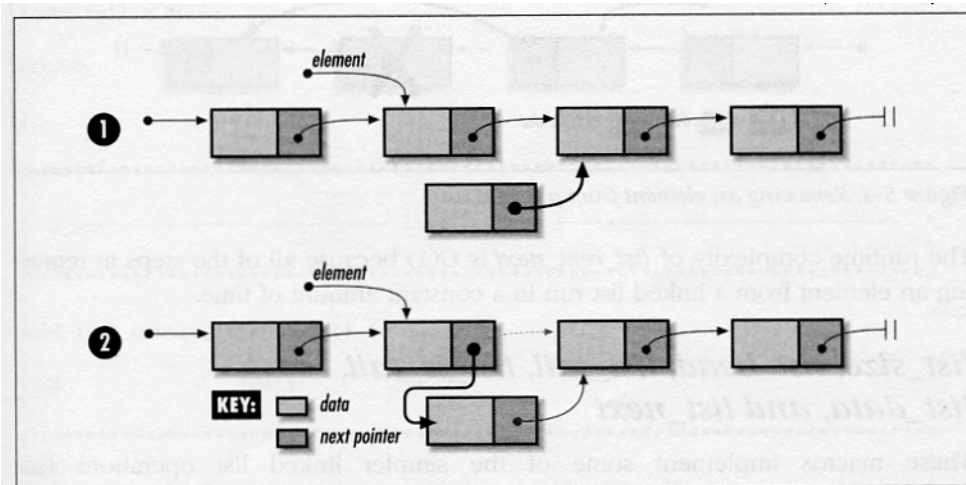


Figure 5-3. Inserting an element into a linked list

## 4 Visualization – Removing

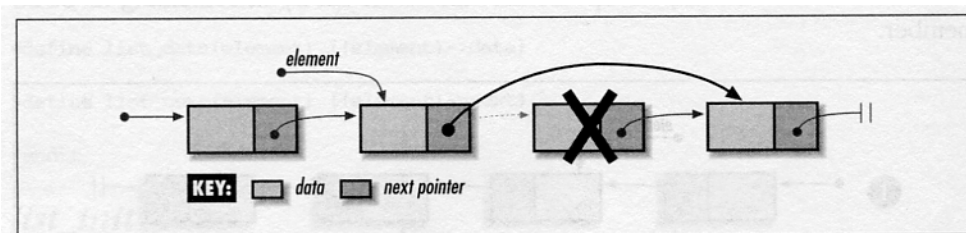


Figure 5-4. Removing an element from a linked list

## 2.3 Singly-Linked List Implementation

### 1 Interface

#### 2 *list.h*

```
3  /*
4   * list.h
5   */
6  #ifndef LIST_H
7  #define LIST_H
8
9  #include <stdlib.h>
10
11 /*
12  * Singly-linked list element
13  */
14 typedef struct ListElmt_
15 {
16     void          *data;
17     struct ListElmt_ *next;
18 } ListElmt;
19
20 /*
21  * Singly-linked list
22  */
23 typedef struct List_
24 {
25     int          size;
26
27     int          (*match)(const void *key1, const void *key2);
28     void          (*destroy)(void *data);
29
30     ListElmt     *head;
31     ListElmt     *tail;
32 } List;
33
34 /*
35  * Public interface
36  */
37 void list_init(List *list, void (*destroy)(void *data));
38
```

```

1 void list_destroy(List *list);
2
3 int list_ins_next(List *list, ListElmt *element, const void *data);
4
5 int list_rem_next(List *list, ListElmt *element, void **data);
6
7 #define list_size(list) ((list)->size)
8
9 #define list_head(list) ((list)->head)
10
11 #define list_tail(list) ((list)->tail)
12
13 #define list_is_head(list, element) ((element) == (list)->head ? 1 : 0)
14
15 #define list_is_tail(element) ((element)->next == NULL ? 1 : 0)
16
17 #define list_data(element) ((element)->data)
18
19 #define list_next(element) ((element)->next)
20
21 #endif

```

- 23 • ListElmt
  - 24 ○ Represents an element in a singly-linked list
  - 25 ○ data
    - 26 ▪ points to data stored in element
  - 27 ○ next
    - 28 ▪ points to next element in list
    - 29 ▪ points to NULL if last element in list
- 30
- 31 • List
  - 32 ○ Represents a singly-linked list
  - 33 ○ size
    - 34 ▪ number of elements in the list
  - 35 ○ match
    - 36 ▪ pointer to function used to compare elements in the list
    - 37 ▪ used by some algorithms (will see this used later in class)
  - 38 ○ destroy



- 1                   ▪ pointer to function called on each data in list when list is
- 2                   destroyed
- 3           ○ head
- 4                   ▪ pointer to first element in list
- 5                   ▪ points to NULL if list is empty
- 6           ○ tail
- 7                   ▪ pointer to last element in list
- 8                   ▪ points to NULL if list is empty
- 9
- 10           • [See pages 53-56 in book]
- 11

## 12 Implementation

### 13 *list.c*

```

14  /*
15   * list.c
16   */
17  #include <stdlib.h>
18  #include <string.h>
19
20  #include "list.h"
21
22  void list_init(List *list, void (*destroy)(void *data))
23  {
24      /* Initialize the list */
25      list->size = 0;
26      list->destroy = destroy;
27      list->head = NULL;
28      list->tail = NULL;
29  }
30
31  void list_destroy(List *list)
32  {
33      void *data;
34
35      /* Remove each element */
36      while (list_size(list) > 0) {
37          if (list_rem_next(list, NULL, (void **)&data) == 0 && list->destroy !=
38              NULL) {
39              /* Call a user-defined function to free dynamically allocated

```

```
1         data. */
2         list->destroy(data);
3     }
4 }
5
6 /* No operations are allowed now, but clear the structure as a
7    precaution. */
8 memset(list, 0, sizeof(List));
9 }
10
11 int list_ins_next(List *list, ListElmt *element, const void *data)
12 {
13     ListElmt *new_element;
14
15     /* Allocate storage for the element. */
16     if ((new_element = (ListElmt *)malloc(sizeof(ListElmt))) == NULL)
17         return -1;
18
19     /* Insert the element into the list. */
20     new_element->data = (void *)data;
21     if (element == NULL) {
22         /* Handle insertion at the head of the list. */
23         if (list_size(list) == 0)
24             list->tail = new_element;
25         new_element->next = list->head;
26         list->head = new_element;
27     }
28     else {
29         /* Handle insertion somewhere other than at the head. */
30         if (element->next == NULL)
31             list->tail = new_element;
32         new_element->next = element->next;
33         element->next = new_element;
34     }
35
36     /* Adjust the size of the list to account for the inserted element. */
37     list->size++;
38
39     return 0;
40 }
41
42 int list_rem_next(List *list, ListElmt *element, void **data)
```

```
1 {
2     ListElmt *old_element;
3
4     /* Do not allow removal from an empty list. */
5     if (list_size(list) == 0)
6         return -1;
7
8     /* Remove the element from the list. */
9     if (element == NULL) {
10         /* Handle removal from the head of the list. */
11         *data = list->head->data;
12         old_element = list->head;
13         list->head = list->head->next;
14
15         if (list_size(list) == 1)
16             list->tail = NULL;
17     }
18     else {
19         /* Handle removal from somewhere other than the head. */
20         if (element->next == NULL)
21             return -1;
22
23         *data = element->next->data;
24         old_element = element->next;
25         element->next = element->next->next;
26
27         if (element->next == NULL)
28             list->tail = element;
29     }
30
31     /* Free the storage allocated by the abstract data type. */
32     free(old_element);
33
34     /* Adjust the size of the list to account for the removed element. */
35     list->size--;
36
37     return 0;
38 }
39
```

## 2.4 Singly-Linked List Example #1

### 1 Inserting numbers into a singly-linked list

```
2  /*
3   * File:  insert-numbers-into-list.c
4   *
5   * Program demonstrating inserting random numbers at tail of singly-linked
6   * list.
7   */
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <time.h>
11
12 #include "list.h"
13
14 int main()
15 {
16     List listOfInts;
17     ListElmt *pElmt;
18     int *pInt;
19     int i;
20
21     /* Seed the random number generator */
22     srand(time(NULL));
23
24     /* Initialize list */
25     list_init(&listOfInts, free);
26
27     /* Output number of elements in list */
28     printf("Before inserts - list size: %d\n", list_size(&listOfInts));
29
30     /* Insert some data at tail of list */
31     for (i = 0; i < 5; ++i) {
32         /* Dynamically allocate an int */
33         pInt = (int *)malloc(sizeof(int));
34         *pInt = random() % 100;
35
36         /* Insert the int at tail of list */
37         printf("Inserting %d into list\n", *pInt);
38         list_ins_next(&listOfInts, list_tail(&listOfInts), pInt);
39     }
```

```
1  /* Output number of elements in list */
2  printf("After inserts - list size: %d\n", list_size(&listOfInts));
3
4  /* Output the data in the list */
5  printf("Elements:");
6  pElmt = list_head(&listOfInts);
7  while (pElmt != NULL) {
8      pInt = (int *)list_data(pElmt);
9      printf(" %d", *pInt);
10     pElmt = list_next(pElmt);
11 }
12 printf("\n");
13
14 /* Destroy the list (this automatically calls destroyInt on each data) */
15 list_destroy(&listOfInts);
16
17 /* Output number of elements in list */
18 printf("After destroy - list size: %d\n", list_size(&listOfInts));
19
20 return EXIT_SUCCESS;
21 }
22
23 /* Program output:
24     Before inserts - list size: 0
25     Inserting 83 into list
26     Inserting 86 into list
27     Inserting 77 into list
28     Inserting 15 into list
29     Inserting 93 into list
30     After inserts - list size: 5
31     Elements: 83 86 77 15 93
32     After destroy - list size: 0
33 */
34
35
36
```

## 2.5 Singly-Linked List Example #2

### 1 Merging two sorted singly-linked lists

```
2  /*
3   * File:  merge-two-sorted-lists.c
4   *
5   * Demonstrate merging two ordered singly-linked lists into
6   * one ordered singly-linked list.
7   */
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11
12 #include "list.h"
13
14 void loadDataIntoList(List *pList, const char *listData[], int listDataLength);
15 void mergeSortedLists(const List *pList1, const List *pList2, List *pDestList);
16 void outputList(List *pList);
17
18 int main()
19 {
20     /* Variables */
21     const char *list1Data[] = {
22         "aardvark",
23         "badger",
24         "horse",
25         "zebra"
26     };
27     const char *list2Data[] = {
28         "armadillo",
29         "baboon",
30         "cat",
31         "kangaroo"
32     };
33     List list1;
34     List list2;
35     List mergedList;
36
37     /* Initialize the lists */
38     list_init(&list1, NULL);
39     list_init(&list2, NULL);
```

```
1  list_init(&mergedList, NULL);
2
3  /* Load sorted data into lists */
4  loadDataIntoList(&list1, list1Data, sizeof(list1Data) / sizeof(*list1Data));
5  loadDataIntoList(&list2, list2Data, sizeof(list2Data) / sizeof(*list2Data));
6
7  /* Merge sorted list into destination list */
8  mergeSortedLists(&list1, &list2, &mergedList);
9
10 /* Output contents of all three lists */
11 outputList(&list1);
12 outputList(&list2);
13 outputList(&mergedList);
14
15 /* Destroy the lists */
16 list_destroy(&list1);
17 list_destroy(&list2);
18 list_destroy(&mergedList);
19
20 return EXIT_SUCCESS;
21 }
22
23 void loadDataIntoList(List *pList, const char *listData[], int listDataLength)
24 {
25     int i;
26
27     for (i = 0; i < listDataLength; ++i)
28     {
29         list_ins_next(pList, list_tail(pList), listData[i]);
30     }
31 }
32
33 void mergeSortedLists(const List *pList1, const List *pList2, List *pDestList)
34 {
35     ListElmt *pList1Elmt;
36     ListElmt *pList2Elmt;
37     const char *pList1Data;
38     const char *pList2Data;
39     const char *pDataToInsert;
40
41     /* Get head element from both source lists */
42     pList1Elmt = list_head(pList1);
```

```
1  pList2Elmt = list_head(pList2);
2
3  /* Add all elements from both lists to the destination list while
4     maintaining sorted order */
5  while (pList1Elmt != NULL || pList2Elmt != NULL)
6  {
7      /* Determine which data should be added next */
8      if (pList1Elmt == NULL)
9      {
10         /* Everything from list 1 has been added, select next from list 2 */
11         pDataToInsert = (const char *)list_data(pList2Elmt);
12         pList2Elmt = list_next(pList2Elmt);
13     }
14     else if (pList2Elmt == NULL)
15     {
16         /* Everything from list 2 has been added, select next from list 1 */
17         pDataToInsert = (const char *)list_data(pList1Elmt);
18         pList1Elmt = list_next(pList1Elmt);
19     }
20     else
21     {
22         /* List 1 and 2 contain more elements */
23         pList1Data = (const char *)list_data(pList1Elmt);
24         pList2Data = (const char *)list_data(pList2Elmt);
25
26         /* Determine whether element from list 1 or 2 should be added next */
27         if (strcmp(pList1Data, pList2Data) < 0)
28         {
29             /* List 1 contains smaller element, select next from list 1 */
30             pDataToInsert = (const char *)list_data(pList1Elmt);
31             pList1Elmt = list_next(pList1Elmt);
32         }
33         else
34         {
35             /* List 2 contains smaller element, select next from list 2 */
36             pDataToInsert = (const char *)list_data(pList2Elmt);
37             pList2Elmt = list_next(pList2Elmt);
38         }
39     }
40
41     /* Add the data to the end of the destination list */
42     list_ins_next(pDestList, list_tail(pDestList), pDataToInsert);
```



```
1     }
2 }
3
4 void outputList(List *pList)
5 {
6     ListElmt *pListElmt;
7
8     printf("List\n");
9     printf("----\n");
10
11     /* Get head element of list */
12     pListElmt = list_head(pList);
13
14     /* Output each element */
15     while (pListElmt != NULL)
16     {
17         printf("%s\n", (const char *)list_data(pListElmt));
18         pListElmt = list_next(pListElmt);
19     }
20
21     printf("\n");
22 }
23
24 /*
25  * Program output:
26      List
27      ----
28      aardvark
29      badger
30      horse
31      zebra
32
33      List
34      ----
35      armadillo
36      baboon
37      cat
38      kangaroo
39
40      List
41      ----
42      aardvark
```

```
1      armadillo
2      baboon
3      badger
4      cat
5      horse
6      kangaroo
7      zebra
8  */
```

## 2.6 Doubly-Linked Lists

### 1 Overview

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

### 7 Visualization – Inserting

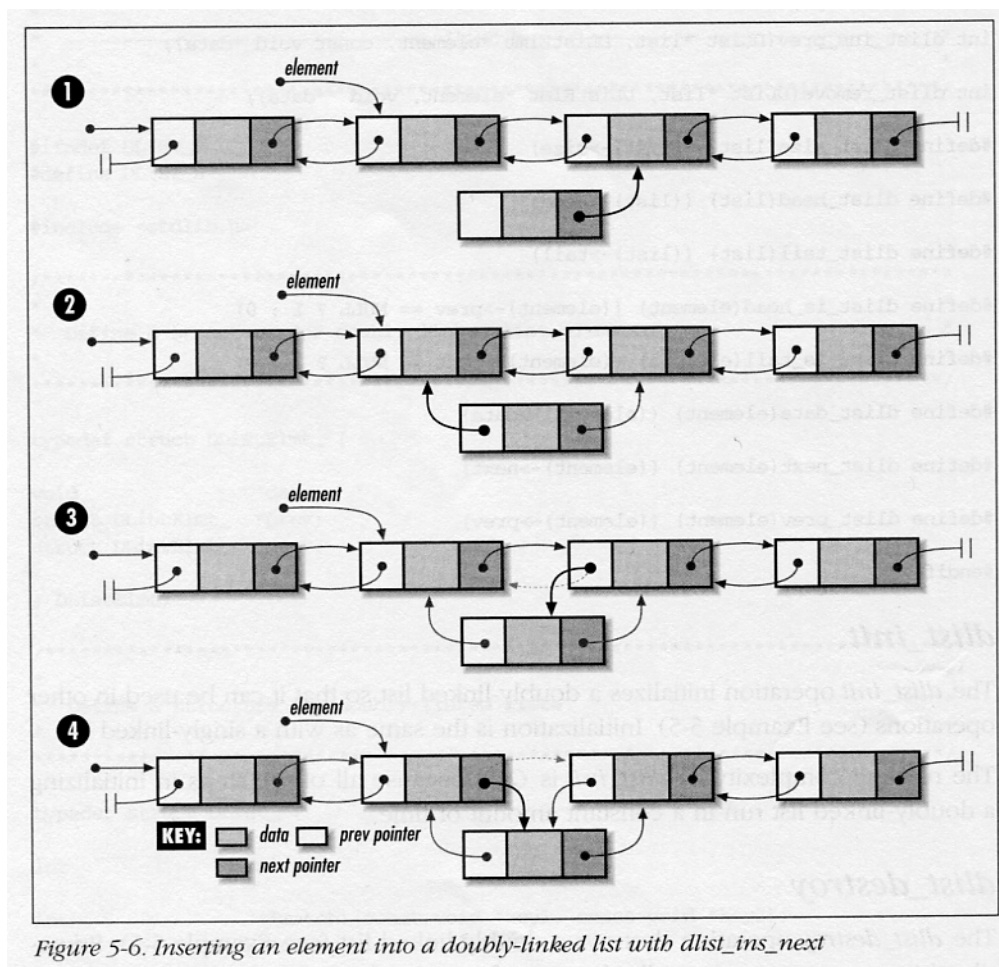


Figure 5-6. Inserting an element into a doubly-linked list with `dlist_ins_next`

## 2.7 Doubly-Linked List Implementation

### 1 Interface

#### 2 *dlist.h*

```
3  /*
4   * dlist.h
5   */
6  #ifndef DLIST_H
7  #define DLIST_H
8
9  #include <stdlib.h>
10
11 /*
12  * Doubly-linked list element
13  */
14 typedef struct DListElmt_
15 {
16     void          *data;
17     struct DListElmt_ *prev;
18     struct DListElmt_ *next;
19 } DListElmt;
20
21 /*
22  * Doubly-linked list
23  */
24 typedef struct DList_
25 {
26     int          size;
27
28     int          (*match)(const void *key1, const void *key2);
29     void          (*destroy)(void *data);
30
31     DListElmt    *head;
32     DListElmt    *tail;
33 } DList;
34
35 /*
36  * Public interface
37  */
38 void dlist_init(DList *list, void (*destroy)(void *data));
```

```
1
2 void dlist_destroy(DList *list);
3
4 int dlist_ins_next(DList *list, DListElmt *element, const void *data);
5
6 int dlist_ins_prev(DList *list, DListElmt *element, const void *data);
7
8 int dlist_remove(DList *list, DListElmt *element, void **data);
9
10 #define dlist_size(list) ((list)->size)
11
12 #define dlist_head(list) ((list)->head)
13
14 #define dlist_tail(list) ((list)->tail)
15
16 #define dlist_is_head(element) ((element)->prev == NULL ? 1 : 0)
17
18 #define dlist_is_tail(element) ((element)->next == NULL ? 1 : 0)
19
20 #define dlist_data(element) ((element)->data)
21
22 #define dlist_next(element) ((element)->next)
23
24 #define dlist_prev(element) ((element)->prev)
25
26 #endif
```

- 28 • DListElmt
  - 29 ○ Represents an element in a doubly-linked list
  - 30 ○ data
    - 31 ▪ points to data stored in element
  - 32 ○ next
    - 33 ▪ points to next element in list
    - 34 ▪ points to NULL if last element in list
  - 35 ○ prev
    - 36 ▪ points to previous element in list
    - 37 ▪ points to NULL if first element in list

- 39 • 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)
```

```
1 {
2     void *data;
3
4     /* Remove each element. */
5     while (dlist_size(list) > 0) {
6         if (dlist_remove(list, dlist_tail(list), (void **)&data) == 0 &&
7             list->destroy != NULL) {
8             /* Call a user-defined function to free dynamically allocated
9              data. */
10            list->destroy(data);
11        }
12    }
13
14    /* No operations are allowed now, but clear the structure as a
15     precaution. */
16    memset(list, 0, sizeof(DList));
17 }
18
19 int dlist_ins_next(DList *list, DListElmt *element, const void *data)
20 {
21     DListElmt *new_element;
22
23     /* Do not allow a NULL element unless the list is empty. */
24     if (element == NULL && dlist_size(list) != 0)
25         return -1;
26
27     /* Allocate storage for the element. */
28     if ((new_element = (DListElmt *)malloc(sizeof(DListElmt))) == NULL)
29         return -1;
30
31     /* Insert the new element into the list. */
32     new_element->data = (void *)data;
33
34     if (dlist_size(list) == 0) {
35         /* Handle insertion when the list is empty. */
36         list->head = new_element;
37         list->head->prev = NULL;
38         list->head->next = NULL;
39         list->tail = new_element;
40     }
41     else {
42         /* Handle insertion when the list is not empty. */
```

```
1     new_element->next = element->next;
2     new_element->prev = element;
3
4     if (element->next == NULL)
5         list->tail = new_element;
6     else
7         element->next->prev = new_element;
8
9     element->next = new_element;
10 }
11
12 /* Adjust the size of the list to account for the inserted element. */
13 list->size++;
14
15 return 0;
16 }
17
18 int dlist_ins_prev(DList *list, DListElmt *element, const void *data)
19 {
20     DListElmt *new_element;
21
22     /* Do not allow a NULL element unless the list is empty. */
23     if (element == NULL && dlist_size(list) != 0)
24         return -1;
25
26     /* Allocate storage to be managed by the abstract data type. */
27     if ((new_element = (DListElmt *)malloc(sizeof(DListElmt))) == NULL)
28         return -1;
29
30     /* Insert the new element into the list. */
31     new_element->data = (void *)data;
32
33     if (dlist_size(list) == 0) {
34         /* Handle insertion when the list is empty. */
35         list->head = new_element;
36         list->head->prev = NULL;
37         list->head->next = NULL;
38         list->tail = new_element;
39     }
40     else {
41         /* Handle insertion when the list is not empty. */
42         new_element->next = element;
```



```
1     new_element->prev = element->prev;
2
3     if (element->prev == NULL)
4         list->head = new_element;
5     else
6         element->prev->next = new_element;
7
8     element->prev = new_element;
9 }
10
11 /* Adjust the size of the list to account for the new element. */
12 list->size++;
13
14 return 0;
15 }
16
17 int dlist_remove(DList *list, DListElmt *element, void **data)
18 {
19     /* Do not allow a NULL element or removal from an empty list. */
20     if (element == NULL || dlist_size(list) == 0)
21         return -1;
22
23     /* Remove the element from the list. */
24     *data = element->data;
25     if (element == list->head) {
26         /* Handle removal from the head of the list. */
27         list->head = element->next;
28
29         if (list->head == NULL)
30             list->tail = NULL;
31         else
32             element->next->prev = NULL;
33     }
34     else {
35         /* Handle removal from other than the head of the list. */
36         element->prev->next = element->next;
37
38         if (element->next == NULL)
39             list->tail = element->prev;
40         else
41             element->next->prev = element->prev;
42     }
```

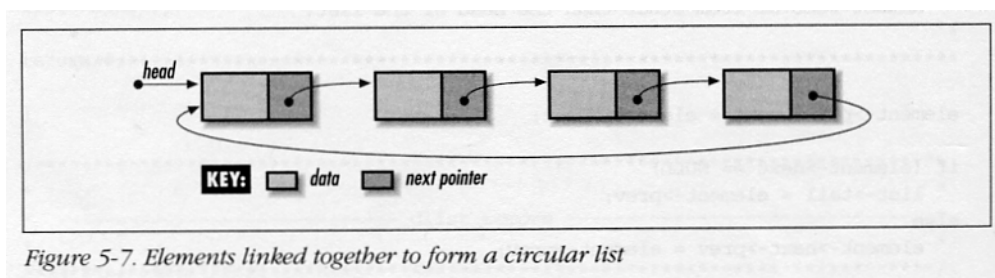
```
1
2  /* Free the storage allocated by the abstract data type. */
3  free(element);
4
5  /* Adjust the size of the list to account for the removed element. */
6  list->size--;
7
8  return 0;
9 }
```

## 2.8 Circular Lists

### 1 Overview

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

### 7 Visualization



## 2.9 Circular List Implementation

### 1 Interface

#### 2 *clist.h*

```
3  /*
4   * clist.h
5   */
6  #ifndef CLIST_H
7  #define CLIST_H
8
9  #include <stdlib.h>
10
11 /*
12  * Circular list element
13  */
14 typedef struct CListElmt_
15 {
16     void          *data;
17     struct CListElmt_ *next;
18 } CListElmt;
19
20 /*
21  * Circular list
22  */
23 typedef struct CList_
24 {
25     int          size;
26
27     int          (*match)(const void *key1, const void *key2);
28     void          (*destroy)(void *data);
29
30     CListElmt    *head;
31 } CList;
32
33 /*
34  * Public interface
35  */
36 void clist_init(CList *list, void (*destroy)(void *data));
37
38 void clist_destroy(CList *list);
```

```
1
2 int clist_ins_next(CList *list, CListElmt *element, const void *data);
3
4 int clist_rem_next(CList *list, CListElmt *element, void **data);
5
6 #define clist_size(list) ((list)->size)
7
8 #define clist_head(list) ((list)->head)
9
10 #define clist_data(element) ((element)->data)
11
12 #define clist_next(element) ((element)->next)
13
14 #endif
```

- 16 • CListElmt
  - 17 ○ Represents an element in a circular linked list
  - 18 ○ data
    - 19 ▪ points to data stored in element
  - 20 ○ next
    - 21 ▪ points to next element in list
    - 22 ▪ points to first element if last element in list
- 24 • CList
  - 25 ○ Represents a circular linked list
  - 26 ○ size
    - 27 ▪ number of elements in the list
  - 28 ○ match
    - 29 ▪ pointer to function used to compare elements in the list
    - 30 ▪ used by some algorithms (will see this used later in class)
  - 31 ○ destroy
    - 32 ▪ pointer to function called on each data in list when list is destroyed
  - 34 ○ head
    - 35 ▪ pointer to first element in list
    - 36 ▪ points to NULL if list is empty

## 1 Implementation

### 2 *clist.c*

```
3  /*
4   * clist.c
5   */
6  #include <stdlib.h>
7  #include <string.h>
8
9  #include "clist.h"
10
11 void clist_init(CList *list, void (*destroy)(void *data))
12 {
13     /* Initialize the list. */
14     list->size = 0;
15     list->destroy = destroy;
16     list->head = NULL;
17 }
18
19 void clist_destroy(CList *list)
20 {
21     void *data;
22
23     /* Remove each element. */
24     while (clist_size(list) > 0) {
25         if (clist_rem_next(list, list->head, (void **)&data) == 0
26             && list->destroy != NULL) {
27             /* Call a user-defined function to free dynamically allocated
28              data. */
29             list->destroy(data);
30         }
31     }
32
33     /* No operations are allowed now, but clear the structure as a
34      precaution. */
35     memset(list, 0, sizeof(CList));
36 }
37
38 int clist_ins_next(CList *list, CListElmt *element, const void *data)
39 {
40     CListElmt *new_element;
```

```
1
2  /* Allocate storage for the element. */
3  if ((new_element = (CListElmt *)malloc(sizeof(CListElmt))) == NULL)
4      return -1;
5
6  /* Insert the element into the list. */
7  new_element->data = (void *)data;
8
9  if (clist_size(list) == 0) {
10      /* Handle insertion when the list is empty. */
11      new_element->next = new_element;
12      list->head = new_element;
13  }
14  else {
15      /* Handle insertion when the list is not empty. */
16      new_element->next = element->next;
17      element->next = new_element;
18  }
19
20  /* Adjust the size of the list to account for the inserted element. */
21  list->size++;
22
23  return 0;
24 }
25
26 int clist_rem_next(CList *list, CListElmt *element, void **data)
27 {
28     CListElmt *old_element;
29
30     /* Do not allow removal from an empty list. */
31     if (clist_size(list) == 0)
32         return -1;
33
34     /* Remove the element from the list. */
35     *data = element->next->data;
36
37     if (element->next == element) {
38         /* Handle removing the last element. */
39         old_element = element->next;
40         list->head = NULL;
41     }
42     else {
```

```
1      /* Handle removing other than the last element. */
2      old_element = element->next;
3      element->next = element->next->next;
4  }
5
6      /* Free the storage allocated by the abstract data type. */
7      free(old_element);
8
9      /* Adjust the size of the list to account for the removed element. */
10     list->size--;
11
12     return 0;
13 }
```