Chapter 14

Dynamic Data Structures

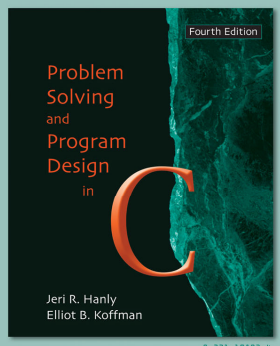

Figure 14.19 Update of List-Traversing Loop Control Variable

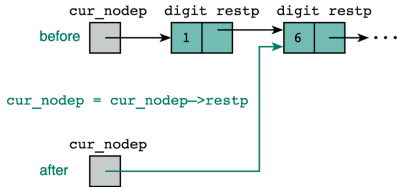

Figure 14.20 Recursive Function get_list

```
1.  #include <stdlib.h>  /*  gives access to malloc */
2.  #define SENT -1
3.  /*
4.   *  Forms a linked list of an input list of integers
5.   *  terminated by SENT
6.   */
7.  list_node_t *
8.  get_list(void)
9.  {
10.       int data;
11.       list_node_t *ansp;
12.
13.       scanf("%d", &data);
14.       if (data == SENT) {
15.             ansp = NULL;
16.       } else {
17.             ansp = (list_node_t *)malloc(sizeof (list_node_t));
18.             ansp->digit = data;
19.             ansp->restp = get_list();
20.       }
21.
22.       return (ansp);
23. }
```


Figure 14.21 Iterative Function get_list

```
1.  /*
2.   *  Forms a linked list of an input list of integers terminated by SENT
3.   */
4.  list_node_t *
5.  get_list(void)
6.  {
7.       int data;
8.       list_node_t *ansp,
9.                   *to_fillp, /* pointer to last node in list whose
10.                              restp component is unfilled       */
11.                  *newp;     /* pointer to newly allocated node   */
12.
13.      /* Builds first node, if there is one */
14.      scanf("%d", &data);
15.      if (data == SENT) {
16.            ansp = NULL;
17.      } else {
18.            ansp = (list_node_t *)malloc(sizeof (list_node_t));
19.            ansp->digit = data;
20.            to_fillp = ansp;
21.
22.            /* Continues building list by creating a node on each
23.               iteration and storing its pointer in the restp component of the
24.               node accessed through to_fillp */
25.            for  (scanf("%d", &data);
26.                  data != SENT;
27.                  scanf("%d", &data)) {
28.                  newp = (list_node_t *)malloc(sizeof (list_node_t));
29.                  newp->digit = data;
30.                  to_fillp->restp = newp;
31.                  to_fillp = newp;
32.            }
33.
34.            /* Stores NULL in final node's restp component */
35.            to_fillp->restp = NULL;
36.      }
37.      return (ansp);
38. }
```


Figure 14.22 Function search

```
1.  /*
2.   *  Searches a list for a specified target value.  Returns a pointer to
3.   *  the first node containing target if found.  Otherwise returns NULL.
4.   */
5.  list_node_t *
6.  search(list_node_t *headp,  /* input - pointer to head of list  */
7.         int         target)  /* input - value to search for      */
8.  {
9.       list_node_t *cur_nodep;  /* pointer to node currently being checked */
10.
11.      for  (cur_nodep = headp;
12.            cur_nodep != NULL  &&  cur_nodep->digit != target;
13.            cur_nodep = cur_nodep->restp) {}
14.
15.       return (cur_nodep);
16. }
```


Figure 14.23 Linked List Representation of Stacks

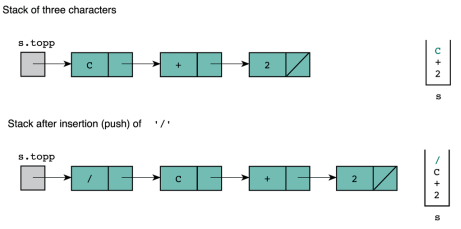

Figure 14.24 Structure Types for a Linked List Implementation of a Stack

```
typedef char stack_element_t;

typedef struct stack_node_s {
      stack_element_t      element;
      struct stack_node_s *restp;
} stack_node_t;

typedef struct {
      stack_node_t *topp;
} stack_t;
```


Figure 14.25 Stack Manipulation with Functions push and pop

```
/*
 *  Creates and manipulates a stack of characters
 */

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

/*  Include typedefs from Fig. 14.24 */
void push(stack_t *sp, stack_element_t c);
stack_element_t pop(stack_t *sp);
```

(continued)

Figure 14.25 Stack Manipulation with Functions push and pop (cont'd)

```
int
main(void)
{
      stack_t s = {NULL};  /* stack of characters - initially empty */

      /*  Builds first stack of Fig. 14.23        */
      push(&s, '2');
      push(&s, '+');
      push(&s, 'C');

      /*  Completes second stack of Fig. 14.23    */
      push(&s, '/');

      /*  Empties stack element by element        */
      printf("\nEmptying stack: \n");
      while (s.topp != NULL) {
            printf("%c\n", pop(&s));
      }

      return (0);
}
/*
 *  The value in c is placed on top of the stack accessed through sp
 *  Pre:  the stack is defined
 */
void
push(stack_t        *sp, /* input/output - stack          */
     stack_element_t c)  /* input       - element to add */
{
      stack_node_t *newp;  /* pointer to new stack node  */

      /*  Creates and defines new node                   */
      newp = (stack_node_t *)malloc(sizeof (stack_node_t));
      newp->element = c;
      newp->restp = sp->topp;
```

(continued)

Figure 14.25 Stack Manipulation with Functions push and pop (cont'd)

```
      /*  Sets stack pointer to point to new node       */
      sp->topp = newp;
}

/*
 *  Removes and frees top node of stack, returning character value
 *  stored there.
 *  Pre:  the stack is not empty
 */
stack_element_t
pop(stack_t *sp) /* input/output - stack */
{
      stack_node_t    *to_freep;  /* pointer to node removed */
      stack_element_t  ans;       /* value at top of stack   */

      to_freep = sp->topp;           /* saves pointer to node being deleted     */
      ans = to_freep->element;       /* retrieves value to return               */
      sp->topp = to_freep->restp;    /* deletes top node                        */
      free(to_freep);                /* deallocates space                       */

      return (ans);
}

Emptying stack:
/
C
+
2
```


Figure 14.26 Structure Types for a Linked List Implementation of a Queue

```
/*  Insert typedef for queue_element_t */

typedef struct queue_node_s {
      queue_element_t      element;
      struct queue_node_s *restp;
} queue_node_t;

typedef struct {
     queue_node_t *frontp,
                  *rearp;
     int           size;
} queue_t;
```


Figure 14.27 A Queue of Passengers in a Ticket Line

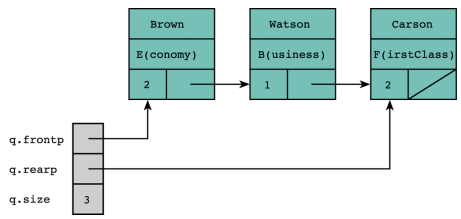

Figure 14.28 Creating and Maintaining a Queue

```
/*
 *  Creates and manipulates a queue of passengers.
 */

int scan_passenger(queue_element_t *passp);
void print_passenger(queue_element_t pass);
void add_to_q(queue_t *qp, queue_element_t ele);
queue_element_t remove_from_q(queue_t *qp);
void display_q(queue_t q);

int
main(void)
{
      queue_t pass_q = {NULL, NULL, 0}; /* passenger queue - initialized to
                                           empty state */
      queue_element_t next_pass, fst_pass;
      char choice;  /* user's request */

      /*  Processes requests  */
      do {
          printf("Enter A(dd), R(emove), D(isplay), or Q(uit)> ");
          scanf(" %c", &choice);
          switch (toupper(choice)) {
          case 'A':
                printf("Enter passenger data> ");
                scan_passenger(&next_pass);
                add_to_q(&pass_q, next_pass);
                break;
```

(continued)

Figure 14.28 Creating and Maintaining a Queue (cont'd)

```
          case 'R':
                if (pass_q.size > 0) {
                      fst_pass = remove_from_q(&pass_q);
                      printf("Passenger removed from queue: \n");
                      print_passenger(fst_pass);
                } else {
                      printf("Queue empty - noone to delete\n");
                }
                break;

          case 'D':
                if (pass_q.size > 0)
                      display_q(pass_q);
                else
                      printf("Queue is empty\n");
                break;

          case 'Q':
                printf("Leaving passenger queue program with %d \n",
                       pass_q.size);
                printf("passengers in the queue\n");
                break;

          default:
                printf("Invalid choice -- try again\n");
          }
      } while (toupper(choice) != 'Q');

      return (0);
}
```


Figure 14.29 Functions add_to_q and remove_from_q

```
/*
 *  Adds ele at the end of queue accessed through qp
 *  Pre:  queue is not empty
 */
void
add_to_q(queue_t        *qp,  /* input/output - queue   */
         queue_element_t  ele) /* input - element to add */
{
      if (qp->size == 0) {              /* adds to empty queue              */
            qp->rearp = (queue_node_t *)malloc(sizeof (queue_node_t));
            qp->frontp = qp->rearp;
      } else {                          /* adds to nonempty queue           */
            qp->rearp->restp =
                  (queue_node_t *)malloc(sizeof (queue_node_t));
            qp->rearp = qp->rearp->restp;
      }
      qp->rearp->element = ele;         /* defines newly added node         */
      qp->rearp->restp = NULL;
      ++(qp->size);
}

/*
 *  Removes and frees first node of queue, returning value stored there.
 *  Pre:  queue is not empty
 */
queue_element_t
remove_from_q(queue_t *qp) /* input/output - queue */
{
      queue_node_t    *to_freep;    /* pointer to node removed              */
      queue_element_t  ans;         /* initial queue value which is to
                                       be returned                          */
```

(continued)

Figure 14.29 Functions add_to_q and remove_from_q (cont'd)

```
      to_freep = qp->frontp;            /* saves pointer to node being deleted */
      ans = to_freep->element;          /* retrieves value to return           */
      qp->frontp = to_freep->restp;     /* deletes first node                  */
      free(to_freep);                   /* deallocates space                   */
      --(qp->size);

      if (qp->size == 0)                /* queue's ONLY node was deleted       */
            qp->rearp = NULL;

      return (ans);
}
```


Figure 14.30 Addition of One Passenger to a Queue

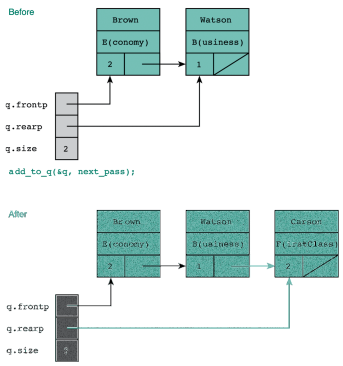

Figure 14.31 Removal of One Passenger from a Queue

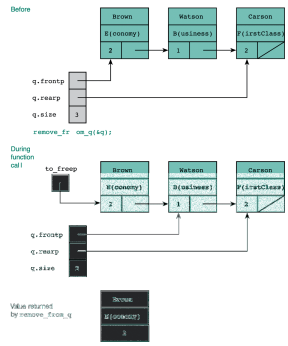

Figure 14.32 Building an Ordered List through Insertions and Deletions

```
/*
 *  Program that builds an ordered list through insertions and then modifies
 *  it through deletions.
 */

typedef struct list_node_s {
        int                 key;
        struct list_node_s *restp;
} list_node_t;
```

(continued)

Figure 14.32 Building an Ordered List through Insertions and Deletions (cont'd)

```
typedef struct {
        list_node_t *headp;
        int          size;
} ordered_list_t;

list_node_t *insert_in_order(list_node_t *old_listp, int new_key);
void insert(ordered_list_t *listp, int key);
int delete(ordered_list_t *listp, int target);
void print_list(ordered_list_t list);

#define SENT -999

int
main(void)
{
        int            next_key;
        ordered_list_t my_list = {NULL, 0};

        /* Creates list through in-order insertions */
        printf("Enter integer keys--end list with %d\n", SENT);
        for  (scanf("%d", &next_key);
              next_key != SENT;
              scanf("%d", &next_key)) {
            insert(&my_list, next_key);
        }

        /* Displays complete list */
        printf("\nOrdered list before deletions:\n");
        print_list(my_list);

        /* Deletes nodes as requested */
        printf("\nEnter a value to delete or %d to quit> ", SENT);
        for  (scanf("%d", &next_key);
              next_key != SENT;
              scanf("%d", &next_key)) {
            if (delete(&my_list, next_key)) {
                printf("%d deleted.  New list:\n", next_key);
                print_list(my_list);
            } else {
                printf("No deletion.  %d not found\n", next_key);
            }

        }
```

(continued)

Figure 14.32 Building an Ordered List through Insertions and Deletions (cont'd)

```
        }

        return (0);
}

Enter integer keys--end list with -999
5  8  4  6  -999
Ordered list before deletions:
   size = 4
   list = 4
          5
          6
          8

Enter a value to delete or -999 to quit> 6
6 deleted.  New list:
   size = 3
   list = 4
          5
          8

Enter a value to delete or -999 to quit> 4
4 deleted.  New list:
   size = 2
   list = 5
          8

Enter a value to delete or -999 to quit> -999
```


Figure 14.33 Cases for Recursive Function insert_in_order

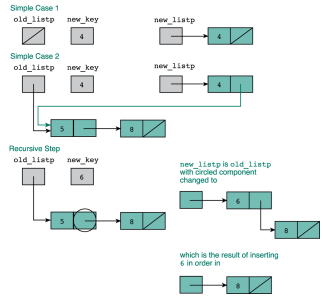

Figure 14.34 Function insert and Recursive Function insert_in_order

```
/*
 *  Inserts a new node containing new_key in order in old_list, returning as
 *  the function value a pointer to the first node of the new list
 */
list_node_t *
insert_in_order(list_node_t *old_listp,  /* input/output */
                int          new_key)    /* input        */
{
        list_node_t *new_listp;

        if (old_listp == NULL) {
              new_listp = (list_node_t *)malloc(sizeof (list_node_t));
              new_listp->key = new_key;
              new_listp->restp = NULL;
        } else if (old_listp->key >= new_key) {
              new_listp = (list_node_t *)malloc(sizeof (list_node_t));
              new_listp->key = new_key;
              new_listp->restp = old_listp;
        } else {
              new_listp = old_listp;
              new_listp->restp = insert_in_order(old_listp->restp, new_key);
        }

        return (new_listp);
}

/*
 *  Inserts a node in an ordered list.
 */
void
insert(ordered_list_t *listp, /* input/output - ordered list */
       int             key)   /* input */
{
        ++(listp->size);
        listp->headp = insert_in_order(listp->headp, key);
}
```


Figure 14.35 Iterative Function delete

```
/*
 *  Deletes first node containing the target key from an ordered list.
 *  Returns 1 if target found and deleted, 0 otherwise.
 */
int
delete(ordered_list_t *listp,      /* input/output - ordered list   */
       int             target)     /* input - key of node to delete */
{
        list_node_t *to_freep,     /* pointer to node to delete                */
                    *cur_nodep;    /* pointer used to traverse list until it
                                      points to node preceding node to delete  */
        int          is_deleted;

        /*  If list is empty, deletion is impossible                           */
        if (listp->size == 0) {
            is_deleted = 0;

        /*  If target is in first node, delete it                              */
        } else if (listp->headp->key == target) {
            to_freep = listp->headp;
            listp->headp = to_freep->restp;
            free(to_freep);
            --(listp->size);
            is_deleted = 1;

        /*  Otherwise, look for node before target node; delete target         */
        } else {
            for  (cur_nodep = listp->headp;
                  cur_nodep->restp != NULL  &&  cur_nodep->restp->key < target;
                  cur_nodep = cur_nodep->restp) ;
            if (cur_nodep->restp != NULL  &&  cur_nodep->restp->key == target) {
                to_freep = cur_nodep->restp;
                cur_nodep->restp = to_freep->restp;
                free(to_freep);
                --(listp->size);
                is_deleted = 1;
            } else {
                is_deleted = 0;
            }
        }

        return (is_deleted);
}
```

Figure 14.36 Function delete Using Recursive Helper Function

```
/*
 *  Deletes first node containing the target key from an ordered list.
 *  Returns 1 if target found and deleted, 0 otherwise.
 */
int
delete(ordered_list_t *listp,  /* input/output - ordered list   */
       int            target)  /* input - key of node to delete */
{
      int is_deleted;

      listp->headp = delete_ordered_node(listp->headp, target,
                                         &is_deleted);
      if (is_deleted)
          --(listp->size);

      return (is_deleted);
}
```


Figure 14.37 Recursive Helper Function delete_ordered_node

```
/*
 *  If possible, deletes node containing target key from list whose first
 *  node is pointed to by listp, returning pointer to modified list and
 *  freeing deleted node.  Sets output parameter flag to indicate whether or
 *  not deletion occurred.
 */
list_node_t *
delete_ordered_node(list_node_t *listp,      /* input/output - list to modify */
                    int          target,     /* input - key of node to delete */
                    int         *is_deletedp) /* output - flag indicating
                                                  whether or not target node
                                                  found and deleted          */
{
    list_node_t *to_freep, *ansp;

    /* if list is empty - can't find target node       - simple case 1       */
    if (listp == NULL) {
        *is_deletedp = 0;
        ansp = NULL;

    /* if first node is the target, delete it          - simple case 2       */
    } else if (listp->key == target) {
        *is_deletedp = 1;
        to_freep = listp;
        ansp = listp->nextp;
        free(to_freep);

    /* if past the target value, give up               - simple case 3       */
    } else if (listp->key > target) {
        *is_deletedp = 0;
        ansp = listp;

    /* in case target node is farther down the list,   - recursive step
       have recursive call modify rest of list and then return list          */
    } else {
        ansp = listp;
        ansp->nextp = delete_ordered_node(listp->nextp, target,
                                          is_deletedp);
    }

    return (ansp);
}
```


Figure 14.38 Binary Trees

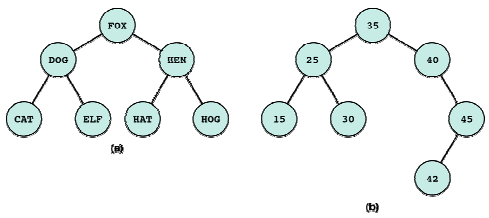

Figure 14.39 Binary Tree Search for 42

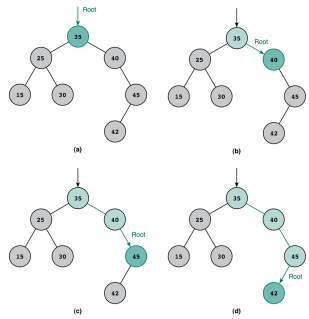

Figure 14.40 Building a Binary Search Tree

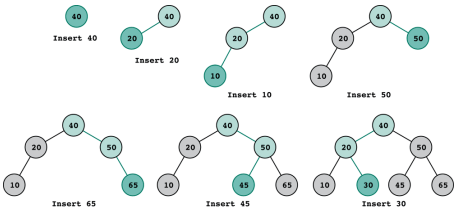

Figure 14.41 Creating a Binary Search Tree

```
/*
 *  Create and display a binary search tree of integer keys.
 */

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

#define TYPED_ALLOC(type) (type *)malloc(sizeof (type))
```

(continued)

Figure 14.41 Creating a Binary Search Tree (cont'd)

```
10. typedef struct tree_node_s {
11.       int               key;
12.       struct tree_node_s *leftp, *rightp;
13. } tree_node_t;
14.
15. tree_node_t *tree_insert(tree_node_t *rootp, int new_key);
16. void tree_inorder(tree_node_t *rootp);
17.
18. int
19. main(void)
20. {
21.       tree_node_t *bs_treep;   /* binary search tree             */
22.       int          data_key;   /* input - keys for tree          */
23.       int          status;     /* status of input operation      */
24.
25.       bs_treep = NULL;   /* Initially, tree is empty */
26.
27.       /* As long as valid data remains, scan and insert keys,
28.          displaying tree after each insertion. */
29.       for  (status = scanf("%d", &data_key);
30.             status == 1;
31.             status = scanf("%d", &data_key)) {
32.          bs_treep = tree_insert(bs_treep, data_key);
33.          printf("Tree after insertion of %d:\n", data_key);
34.          tree_inorder(bs_treep);
35.       }
36.
37.       if (status == 0) {
38.            printf("Invalid data >>%c\n", getchar());
39.       } else {
40.            printf("Final binary search tree:\n");
41.            tree_inorder(bs_treep);
42.       }
43.
44.       return (0);
45. }
46.
```

(continued)

Figure 14.41 Creating a Binary Search Tree (cont'd)

```
47. /*
48.  *  Insert a new key in a binary search tree.  If key is a duplicate,
49.  *  there is no insertion.
50.  *  Pre:  rootp points to the root node of a binary search tree
51.  *  Post: Tree returned includes new key and retains binary
52.  *        search tree properties.
53.  */
54. tree_node_t *
55. tree_insert(tree_node_t *rootp,   /* input/output - root node of
56.                                     binary search tree    */
57.             int          new_key) /* input - key to insert */
58. {
59.       if (rootp == NULL) {              /* Simple Case 1 - Empty tree   */
60.            rootp = TYPED_ALLOC(tree_node_t);
61.            rootp->key = new_key;
62.            rootp->leftp = NULL;
63.            rootp->rightp = NULL;
64.       } else if (new_key == rootp->key) {            /* Simple Case 2 */
65.            /* duplicate key - no insertion                          */
66.       } else if (new_key < rootp->key) {             /* Insert in     */
67.            rootp->leftp = tree_insert                /* left subtree  */
68.                           (rootp->leftp, new_key);
69.       } else {                          /* Insert in right subtree */
70.            rootp->rightp = tree_insert(rootp->rightp,
71.                                        new_key);
72.       }
73.
74.       return (rootp);
75. }
```