Dynamic Data Structures with C

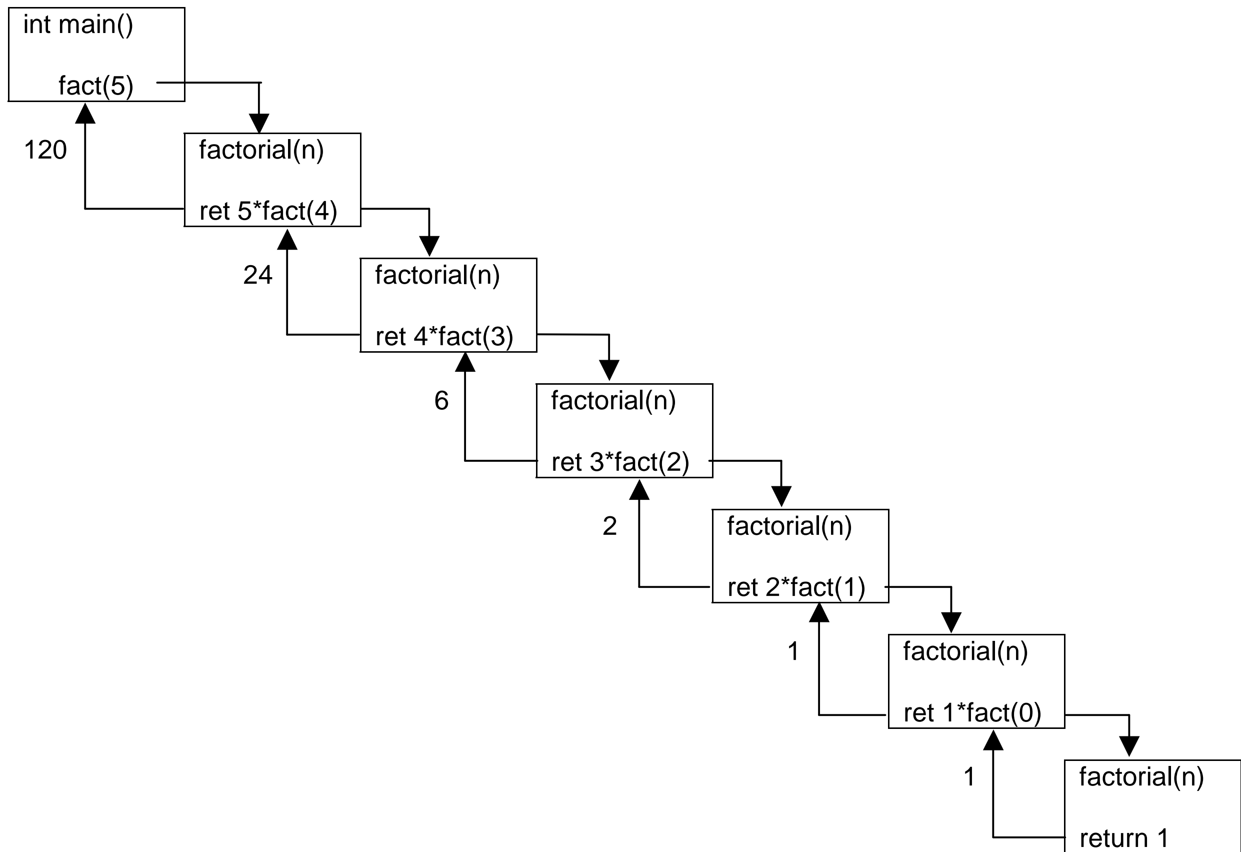
Terry Marris August 2010

Answers

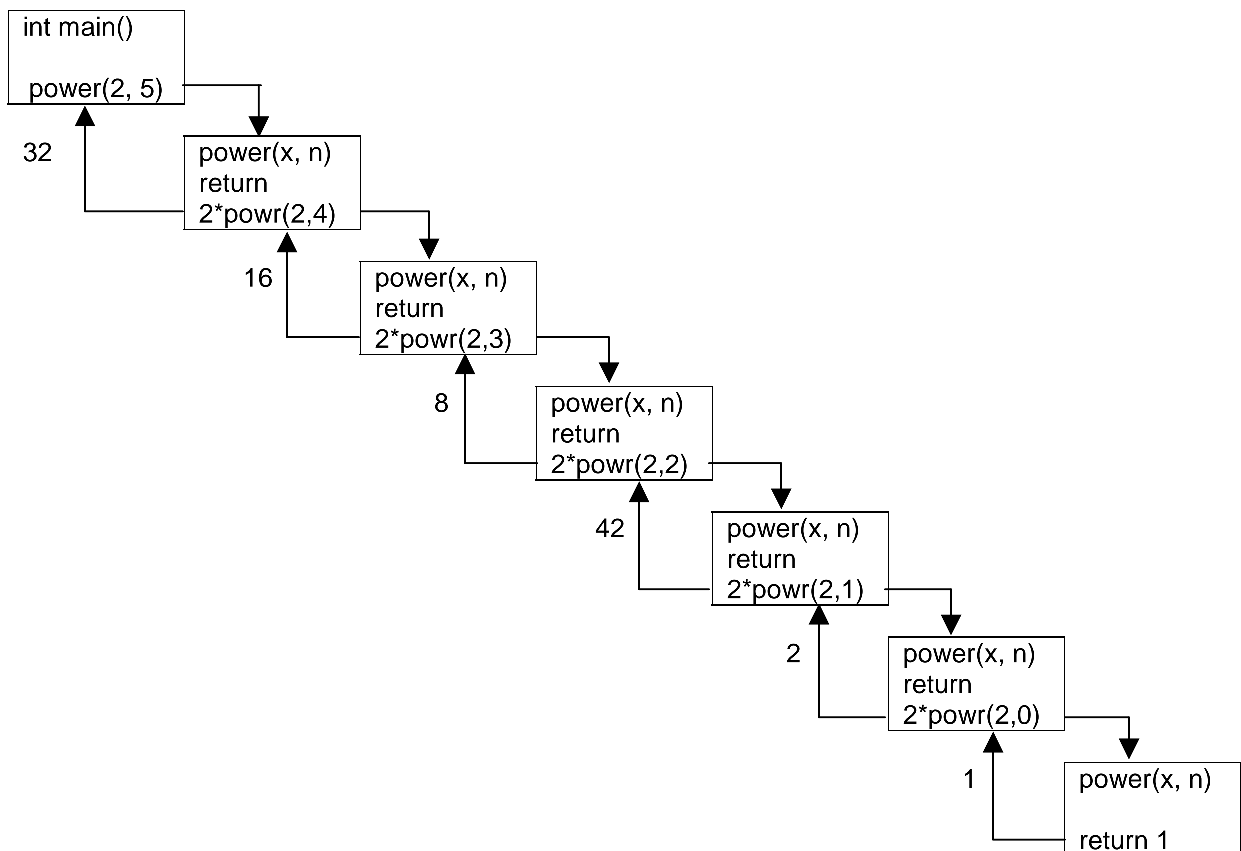
Exercise 1.1

```
/* pointers.c */

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

typedef int (*PtrToFunction)(char, char);

typedef struct structure {
  PtrToFunction ptof;
} *PtrToStructure, Structure;

/* compare: returns 1 if the two given chars are identical */
int compare(const char a, const char b)
{
  if (a == b)
    return 1;
  return 0;
}

/* equal: returns 1 if the two char parameters are equal */
int equal(PtrToFunction ptof, const char a, const char b)
{
  return (*ptof)(a, b);
}

int main()
{
  PtrToFunction ptof = compare;
  printf("Expect not identical: ");
  if (equal((int (*)(char, char))(*ptof), 'a', 'A') == 1)
    printf("identical");
  else
    printf("not identical");
  printf("\n");
  printf("Expect identical: ");
  if (equal((int (*)(char, char))(*ptof), 'a', 'a') == 1)
    printf("identical");
  else
    printf("not identical");
  printf("\n");

  Structure s;
  s.ptof = compare;
  printf("Expect not identical: ");
  if (s.ptof('a', 'A'))
    printf("identical");
  else
    printf("not identical");
  printf("\n");

  PtrToStructure ps = malloc(sizeof(struct structure));
  ps->ptof = compare;
  printf("Expect identical: ");
  if (ps->ptof('a', 'a'))
    printf("identical");
  else
    printf("not identical");
  return 0;
}
```

Exercise 2.1

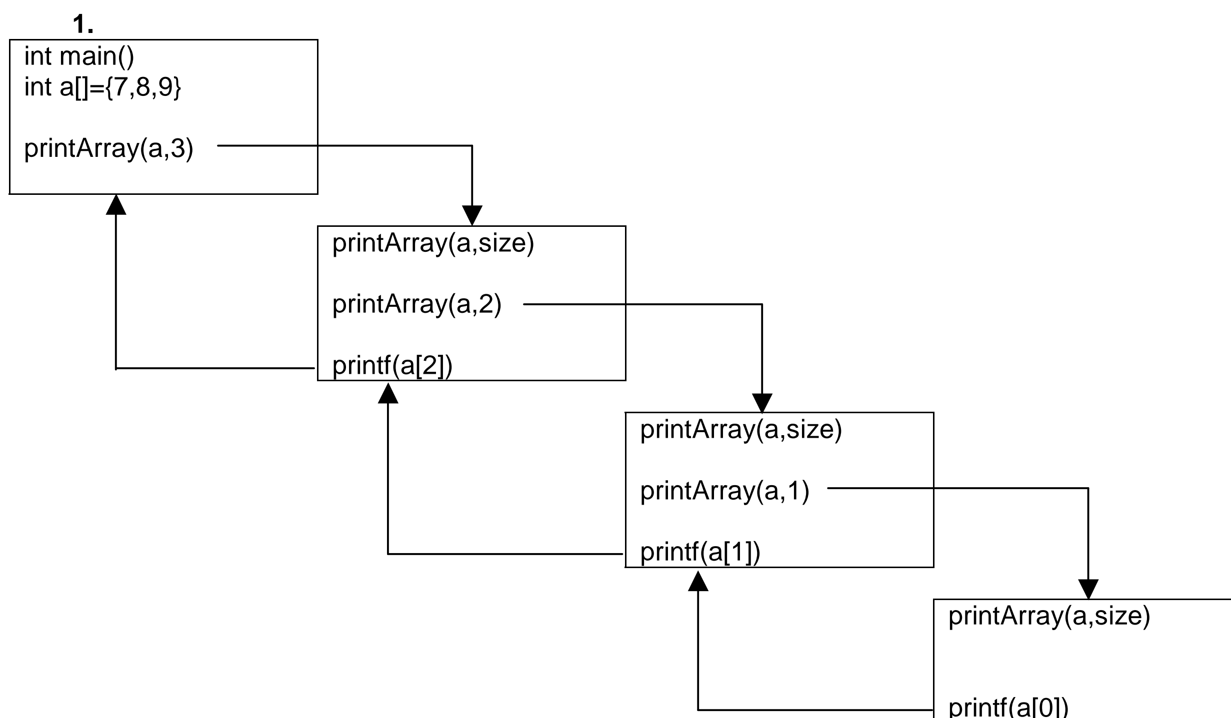


Exercise 2.2



Exercise 2.3

1.



Notice how the *printf()* statements are executed on returning from a call to *printArray()*.

2.

Start with an example array: array =

0	1	2
10	**20**	**30**

Write the first call: *sortArray(array, 3)*

Establish the stopping condition: *arraySize == 1*

Write the function call: *sumArray(array, arraySize-1)*

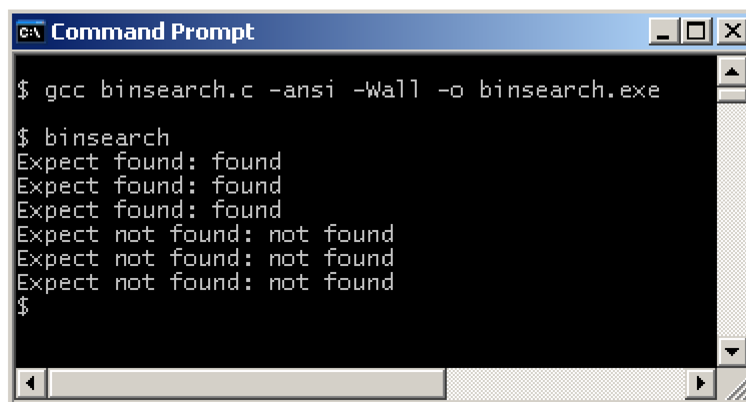
Write the function. (You could sketch the sequence of function calls and their returns if that helps.)

```
/* sumArray.c */

#include <stdio.h>

int sumArray(int array[], int arraySize)
{
  if (arraySize == 1)
    return array[0];
  else
    return array[arraySize-1] + sumArray(array, arraySize-1);
}


int main()
{
  int array[] = { 10, 20, 30 };
  printf("%d", sumArray(array, 3));
  return 0;
}
```

3.

```
/* binsearch.c */

#include <stdio.h>

/* binarySearch: returns 1 if target is found in the sorted array */
int binarySearch(const int array[], int from, int to, const int target)
{
  if (from > to)
    return 0;                                /* stopping case: not found */
  int middle = from + (to - from) / 2;               /* prevent overflow */
  if (target == array[middle])
    return 1;                                   /* stopping case: found */
  else if (target < array[middle])                          /* go left */
    return binarySearch(array, from, middle - 1, target); /*recursive call*/
  else if (target > array[middle])                         /* go right */
    return binarySearch(array, middle + 1, to, target);   /*recursive call*/
  return 0;                                  /* this should not happen */
}

int main()
{
  int array[] = { 10, 20, 30, 40, 50 };            /* array must be sorted */

  printf("Expect found: ");
  binarySearch(array, 0, 4, 10)? printf("found") : printf("not found");
  printf("\n");
  printf("Expect found: ");
  binarySearch(array, 0, 4, 50)? printf("found") : printf("not found");
  printf("\n");
  printf("Expect found: ");
  binarySearch(array, 0, 4, 30)? printf("found") : printf("not found");
  printf("\n");

  printf("Expect not found: ");
  binarySearch(array, 0, 4, 9)? printf("found") : printf("not found");
  printf("\n");
  printf("Expect not found: ");
  binarySearch(array, 0, 4, 51)? printf("found") : printf("not found");
  printf("\n");
  printf("Expect not found: ");
  binarySearch(array, 0, 4, 25)? printf("found") : printf("not found");
  return 0;
}
```

Call the binary search function and provide the sorted array along with its lowest and highest index values, and the target to search for. Each recursive call reduces the range to search through by half until either the target is found, or the range cannot be reduced any further. Naive calculation of the middle index *(from + to) / 2* could result in overflow if *from* and *to* were sufficiently large.

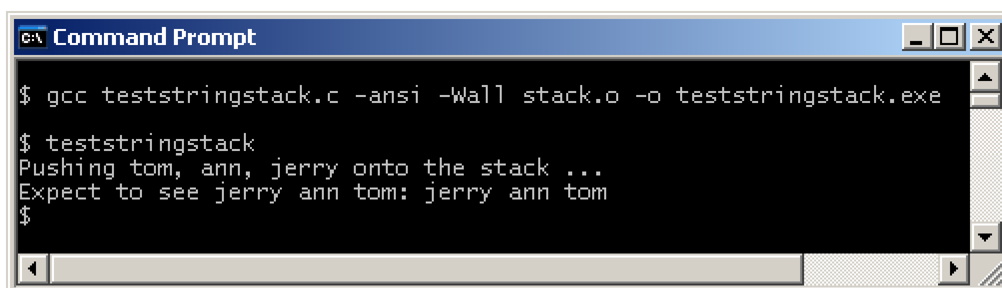


Exercise 3.1

1.
```
/* stackLength: returns the number of nodes in the stack */
int stackLength(const Stack s)
{
  int len = 0;
  StackNode p = s->head;
  while (p != NULL) {
    p = p->next;
    len++;
  }
  return len;
}
```

   StackNode pointer is marched along the nodes, adding one to a counter for each node as it does so.

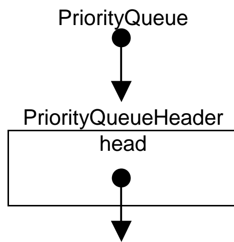
2.
```
/* TestStringStack.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include "stack.h"

/* copyString: returns a duplicate copy of its parameter */
char *copyString(const char *s)
{
  char *t = malloc(strlen(s)+1);
  assert(t != NULL);
  strcpy(t, s);
  return t;
}

int main()
{
  Stack s = newStack((Duplicate)copyString);
  printf("Pushing tom, ann, jerry onto the stack ... \n");

  s = push("tom", s);
  s = push("ann", s);
  s = push("jerry", s);

  printf("Expect to see jerry ann tom: ");
  StackIterator iter = newStackIterator(s);
  while (stackIteratorHasNext(iter))
    printf("%s ", (char *)stackIteratorNext(iter));

  return 0;
}
```



Exercise 4.1

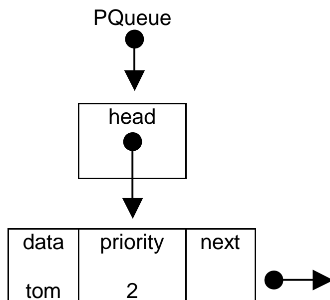
1.
```
/* queueLength: returns the number of data items in the queue */
int queueLength(const Queue q)
{
  int len = 0;
  QueueNode p = q->head;
  while (p != NULL) {
    p = p->next;
    len++;
  }
  return len;
}
```

2. New Priority Queue



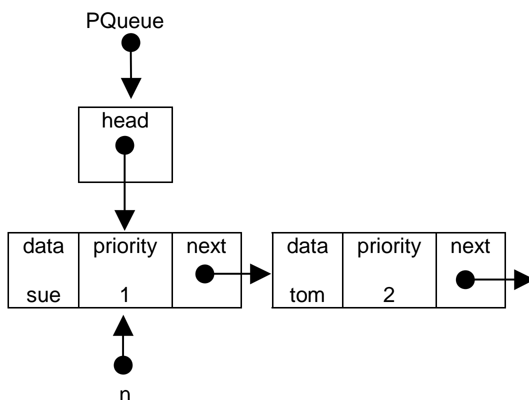

```
PriorityQueue q = (PriorityQueue)malloc(sizeof(
                                  struct PriorityQueueHeader));
q->head = NULL;
```

Case 1: Empty Queue



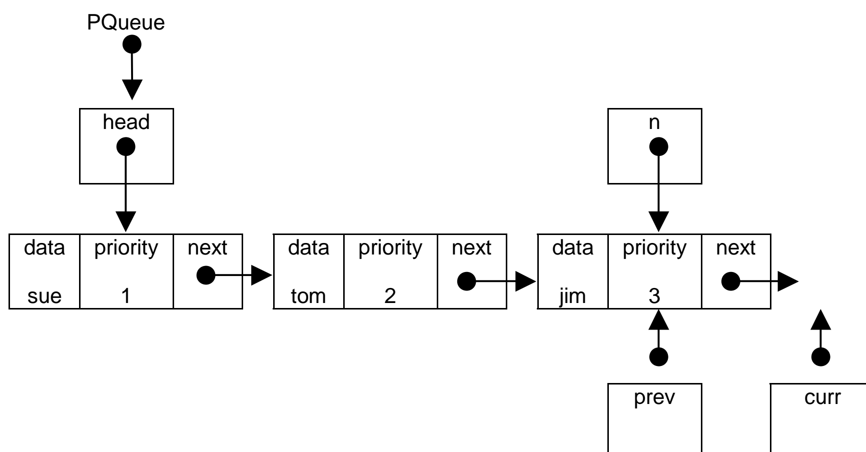

```
PriorityQueueNode curr = NULL;
PriorityQueueNode prev = NULL;
if (q == NULL)
  priorityQueueError("joinPriorityQueue: priority queue is null");
PriorityQueueNode n = newPriorityQueueNode();
n->data = data;
n->priority = priority;
```

Case 2: priority < first node priority



```
if (priority < q->head->priority) {
  n->next = q->head;
 q->head = n;
```

Case 3: priority >= first node priority



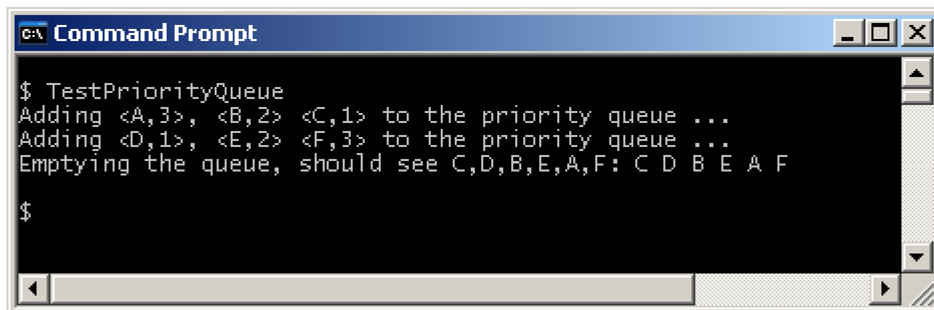

```
  prev = NULL;
  curr = q->head;
  while ((curr != NULL) && (curr->priority <= priority)) {
    prev = curr;
    curr = curr->next;
  }
  prev->next = n;
  if (curr != NULL)
    n->next = curr;
```

```
/* PriorityQueue.h */
#ifndef PQUEUE
#define PQUEUE

typedef struct PriorityQueueHeader *PriorityQueue;

/* priorityQueueError: reports priority queue errors, halts program
   execution */
int priorityQueueError(char *e);

/* newPriorityQueue: returns a new, empty priority queue */
PriorityQueue newPriorityQueue();

/* disposePriorityQueue: deallocates memory occupied by priority
   queue */
PriorityQueue disposePriorityQueue(PriorityQueue *q);

/* joinPriorityQueue: adds data item to a priority queue in order of
   priority, 1 highest, 3 lowest */
PriorityQueue joinPriorityQueue(void *data,
                                        int priority, PriorityQueue q);

/* headOfPriorityQueue: returns item at head of a priority queue,
   leaves queue unchanged */
void *headOfPriorityQueue(const PriorityQueue q);

/* leavePriorityQueue: removes data item from front of priority queue
*/
PriorityQueue leavePriorityQueue(PriorityQueue q);

/* isEmptyPriorityQueue: returns 0 if priority queue is empty */
int isEmptyPriorityQueue(const PriorityQueue q);

#endif
```

```
/* priorityqueue.c */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "priorityqueue.h"

typedef struct PriorityQueueNodeStruct *PriorityQueueNode;

struct PriorityQueueNodeStruct {
  void *data;
  int priority;
  PriorityQueueNode next;
};

struct PriorityQueueHeader {
  PriorityQueueNode head;
};

/* priorityQueueError: reports priority queue errors, halts program
   execution */
int priorityQueueError(char *error)
{
  printf("%s\n", error);
  exit(1);
}

/* newPriorityQueue: returns a new, empty priority queue */
PriorityQueue newPriorityQueue()
{
  PriorityQueue q = (PriorityQueue)malloc(sizeof(
                                      struct PriorityQueueHeader));
  if (q == NULL)
    priorityQueueError("newPriorityQueue: out of memory");
  q->head = NULL;
  return q;
}

/* isEmptyPriorityQueue: returns 0 (true) if priority queue has no elements
*/
int isEmptyPriorityQueue(const PriorityQueue q)
{
  if (q == NULL)
    priorityQueueError("isEmptyPriorityQueue: queue is null");
  return q->head == NULL;
}

/* newPriorityQueueNode: returns a new, empty priority queue node */
PriorityQueueNode newPriorityQueueNode()
{
  PriorityQueueNode n = (PriorityQueueNode)malloc(sizeof(struct
PriorityQueueNodeStruct));
  if (n == NULL)
    priorityQueueError("newPriorityQueueNode: out of memory");
  n->next = NULL;
  n->priority = 0;
  n->data = NULL;
  return n;
}

/* joinPriorityQueue: adds data item in its priority position in a priority
   queue  */
PriorityQueue joinPriorityQueue(void *data, int priority, PriorityQueue q)
{
  PriorityQueueNode curr = NULL;
  PriorityQueueNode prev = NULL;
  if (q == NULL)
    priorityQueueError("joinPriorityQueue: priority queue is null");
  PriorityQueueNode n = newPriorityQueueNode();
  n->data = data;
  n->priority = priority;
```

```
  /* case 1: empty queue */
  if (isEmptyPriorityQueue(q)) {
    q->head = n;
    return q;
  }

  /* case 2: given priority less than priority of first node */
  if (priority < q->head->priority) {
    n->next = q->head;
    q->head = n;
    return q;
  }

  /* case 3: given priority not less than priority of first node */
  prev = NULL;
  curr = q->head;
  while ((curr != NULL) && (curr->priority <= priority)) {
    prev = curr;
    curr = curr->next;
  }
  prev->next = n;
  if (curr != NULL)
    n->next = curr;
  return q;
}

/* headOfPriorityQueue: returns item at head of priority queue */
void *headOfPriorityQueue(const PriorityQueue q)
{
  if (q == NULL)
    priorityQueueError("headOfPriorityQueue: queue is null");
  if (isEmptyPriorityQueue(q))
    priorityQueueError("headOfPriorityQueue: queue is empty");
  return q->head->data;
}

/* disposePriorityQueueNode: releases memory occupied by PriorityQueueNode
*/
PriorityQueueNode disposePriorityQueueNode(PriorityQueueNode n)
{
  free(n);
  n = NULL;
  return n;
}

/* leavePriorityQueue: removes data item from front of priority queue */
PriorityQueue leavePriorityQueue(PriorityQueue q)
{
  if (q == NULL)
    priorityQueueError("leavePriorityQueue: queue is null");
  if (isEmptyPriorityQueue(q))
    priorityQueueError("leavePriorityQueue: queue is empty");
  PriorityQueueNode tail = q->head->next;
  disposePriorityQueueNode(q->head);
  q->head = tail;
  return q;
}

/* disposePriorityQueue: deallocates memory occupied by queue */
PriorityQueue disposePriorityQueue(PriorityQueue *q)
{
  if (*q == NULL)
    priorityQueueError("disposePriorityQueue: queue is null");
  while (!isEmptyPriorityQueue(*q))
    leavePriorityQueue(*q);
  free(*q);
  *q = NULL;
  return *q;
}
```

```
/* TestPriorityQueue */

#include <stdio.h>
#include "priorityqueue.h"

int main()
{
  PriorityQueue q = newPriorityQueue();

  printf("Adding <A,3>, <B,2> <C,1> to the priority queue ... \n");
  char chA = 'A';
  char chB = 'B';
  char chC = 'C';
  joinPriorityQueue(&chA, 3, q);
  joinPriorityQueue(&chB, 2, q);
  joinPriorityQueue(&chC, 1, q);

  printf("Adding <D,1>, <E,2> <F,3> to the priority queue ... \n");
  char chD = 'D';
  char chE = 'E';
  char chF = 'F';
  joinPriorityQueue(&chD, 1, q);
  joinPriorityQueue(&chE, 2, q);
  joinPriorityQueue(&chF, 3, q);

  printf("Emptying the queue, should see C,D,B,E,A,F: ");
  while (!isEmptyPriorityQueue(q)) {
    printf("%c ", *(char *)headOfPriorityQueue(q));
    leavePriorityQueue(q);
  }
  printf("\n");
  return 0;
}
```



Exercise 4.1

1.

```
/* list.h */

#ifndef LIST
#define LIST

typedef struct ListHeader *List;
typedef struct ListIteratorStruct *ListIterator;

/* listError: reports list errors, halts program execution */
int listError(char *e);

/* newList: returns a new list */
List newList();

/* disposeList: deallocates memory occupied by list */
List disposeList(List *list);

/* isEmptyList: returns true if list is empty */
int isEmptyList(const List list);
```

```
/* listSize: returns the number of items in the list */
int listSize(List list);

/* addFirst: inserts data item at front of list */
int addFirst(void *data, List list);

/* addLast: appends data item onto end of list */
int addLast(void *data, List list);

/* getFirst: returns item at front of list */
void *getFirst(const List list);

/* getLast: returns item at end of list */
void *getLast(const List list);

/* removeFirst: removes first item in list */
void *removeFirst(List list);

/* removeLast: removes last item in list */
void *removeLast(List list);

/* newListIterator: returns a new iterator for the given list. */
ListIterator newListIterator(List list);

/* listIteratorHasNext: returns true if there is a node after
   the current iterator position */
int listIteratorHasNext(ListIterator iter);

/* listIteratorHasPrevious: returns true if there is a node before
   the current iterator position */
int listIteratorHasPrevious(ListIterator iter);

/* listIteratorNext: returns the next item */
void *listIteratorNext(ListIterator iter);

/* listIteratorPrevious: returns the previous item */
void *listIteratorPrevious(ListIterator iter);

/* listIteratorAdd: inserts item before the current iterator position */
List listIteratorAdd(void *data, ListIterator iter);

/* listIteratorRemove: removes the last visited item */
List listIteratorRemove(ListIterator iter);

/* disposeListIterator: deallocates memory occupied by
   ListIterator */
ListIterator disposeListIterator(ListIterator *iter);

#endif
```

/* list.c */

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "list.h"
typedef struct ListNodeStruct *ListNode;

struct ListNodeStruct {
  void *data;
  ListNode prior;
  ListNode next;
};

struct ListHeader {
  ListNode head;
  ListNode last;
};

struct ListIteratorStruct {
```

```
  ListNode previous;
  ListNode current;
  List list;
};

/* listError: reports list errors, halt program execution */
int listError(char *error)
{
  printf("\n%s\n", error);
  exit(1);
}

/* newList: returns a new list */
List newList()
{
  List list = (List)malloc(sizeof(struct ListHeader));
  if (list == NULL)
    listError("newList: out of memory");
  list->head = NULL;
  list->last = NULL;
  return list;
}

/* isEmptyList: returns true if list is empty */
int isEmptyList(const List list)
{
  if (list == NULL)
    listError("isEmptyList: list is NULL");
  return (list->head == NULL);
}

/* newListNode: returns a new list node */
ListNode newListNode()
{
  ListNode n = (ListNode)malloc(sizeof(struct ListNodeStruct));
  if (n == NULL)
    listError("listNode: out of memory");
  n->prior = NULL;
  n->next = NULL;
  n->data = NULL;
  return n;
}

/* disposeListNode: deallocates the given node */
ListNode disposeListNode(ListNode n)
{
  free(n);
  n = NULL;
  return n;
}

/* addFirst: inserts data item at front of list */
int addFirst(void *data, List list)
{
  if (list == NULL)
    listError("addFirst: list is NULL");
  ListNode n = newListNode();
  n->data = data;
  if (isEmptyList(list)) {
    list->head = n;
    list->last = n;
  }
  else {
    n->next = list->head;
    list->head->prior = n;
    list->head = n;
  }
  return 0;
}

/* addLast: appends data item onto end of list */
```

```
int addLast(void *data, List list)
{
  if (list == NULL)
    listError("addLast: list is NULL");
  ListNode n = newListNode();
  n->data = data;
  if (isEmptyList(list)) {
    list->head = n;
    list->last = n;
  }
  else {
    n->prior = list->last;
    list->last->next = n;
    list->last = n;
  }
  return 0;
}

/* getFirst: returns item at front of list */
void *getFirst(const List list)
{
  if (list == NULL)
    listError("getFirst: list is NULL");
  if (isEmptyList(list))
    listError("getFirst: list is empty");
  return list->head->data;
}

/* getLast: returns item at end of list */
void *getLast(const List list)
{
  if (list == NULL)
    listError("getLast: list is NULL");
  if (isEmptyList(list))
    listError("getLast: list is empty");
  return list->last->data;
}

/* removeFirst: removes first item in list */
void *removeFirst(List list)
{
  if (list == NULL)
    listError("removeFirst: list is NULL");
  if (isEmptyList(list))
    listError("removeFirst: list is empty");
  ListNode n = list->head;
  void *data = n->data;
  if (list->head->next == NULL) {
    list->head = NULL;
    list->last = NULL;
  }
  else {
    list->head = list->head->next;;
    list->head->prior = NULL;
  }
  disposeListNode(n);
  return data;
}

/* removeLast: removes last item in list */
void *removeLast(List list)
{
  if (list == NULL)
    listError("removeLast: list is NULL");
  if (isEmptyList(list))
    listError("removeLast: list is empty");
  ListNode n = list->last;
  void *data = n->data;
  if (list->last->prior == NULL) {
    list->head = NULL;
    list->last = NULL;
```

```
  }
  else {
    list->last = list->last->prior;
    list->last->next = NULL;
  }
  disposeListNode(n);
  return data;
}

/* listSize: returns the number of items in the list */
int listSize(List list)
{
  if (list == NULL)
    listError("listSize: list is NULL");
  int i = 0;
  ListNode p = list->head;
  while (p != NULL) {
    p = p->next;
    i++;
  }
  return i;
}

/* newListIterator: returns a new list iterator. */
ListIterator newListIterator(List list)
{
  ListIterator iter = (ListIterator)malloc(sizeof(struct
ListIteratorStruct));
  if (iter == NULL)
    listError("newListIterator: out of memory");
  iter->previous = NULL;
  iter->current = NULL;
  iter->list = list;
  return iter;
}

/* listIteratorHasNext: returns true if there is a node after the
   current iterator position */
int listIteratorHasNext(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorHasNext: list is NULL");
  if (isEmptyList(iter->list))
    listError("listIteratorHasNext: list is empty");
  if (iter->current == NULL)
    return iter->list->head != NULL;
  return iter->current->next != NULL;
}

/* listIteratorHasPrevious: returns true if there is a node before
   current iterator position */
int listIteratorHasPrevious(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorHasPrevious: list is NULL");
  if (isEmptyList(iter->list))
    listError("listIteratorHasPrevious: list is empty");
  if (iter->current == NULL)
    return iter->list->head != NULL;
  return iter->current->prior != NULL;
}

/* listIteratorNext: returns the next item */
void *listIteratorNext(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorNext: list is NULL\n");
  if (isEmptyList(iter->list))
    listError("listIteratorNext: list is empty");
  if (!listIteratorHasNext(iter))
    listError("listIteratorNext: no next node");
```

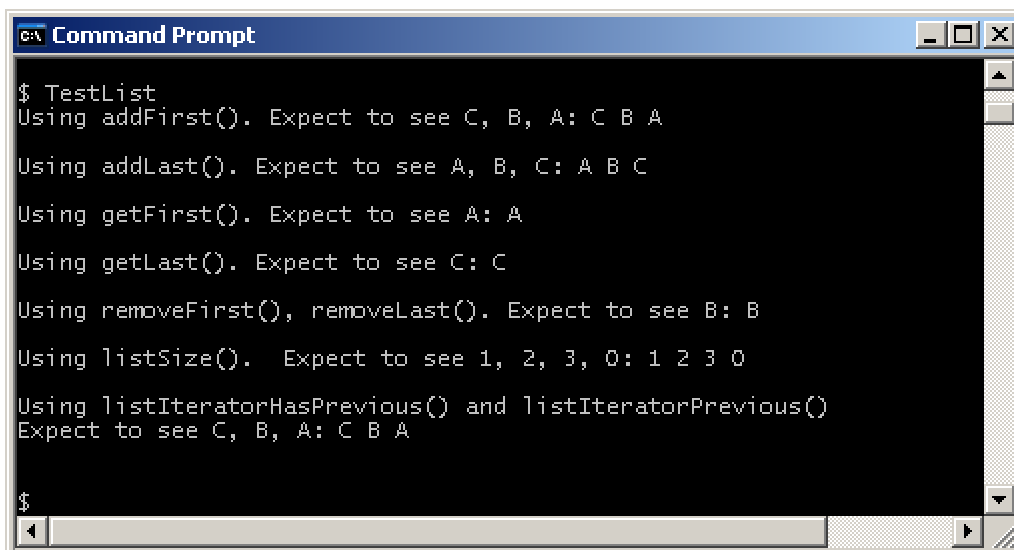
```
  iter->previous = iter->current;
  if (iter->current == NULL)
    iter->current = iter->list->head;
  else
    iter->current = iter->current->next;
  return iter->current->data;
}

/* listIteratorPrevious: returns the previous item */
void *listIteratorPrevious(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorPrevious: list is NULL\n");
  if (isEmptyList(iter->list))
    listError("listIteratorPrevious: list is empty");
  if (!listIteratorHasPrevious(iter))
    listError("listIteratorPrevious: no previous node");
  iter->previous = iter->current;
  if (iter->current == NULL)
    iter->current = iter->list->last;
  else
    iter->current = iter->current->prior;
  return iter->current->data;
}

/* listIteratorAdd: inserts item before the current iterator position */
List listIteratorAdd(void *data, ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorAddItem: list is NULL");
  ListNode n = newListNode();
  n->data = data;
 if (iter->list->head == NULL) {
    iter->list->head = n;
    iter->list->last = n;
  }
  else if (iter->current == NULL) {
    n->next = iter->list->head;
    iter->list->head = n;
  }
  else {
    n->next = iter->current;
    if (iter->list->head == iter->current)
      iter->list->head = n;
    else if (iter->previous != NULL)
      iter->previous->next = n;
  }
  iter->previous = NULL;
  return iter->list;
}

/* listIteratorRemove: removes the last visited item */
List listIteratorRemove(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorRemove: list is NULL");
  if (isEmptyList(iter->list))
    listError("listIteratorRemove: list is empty");
  ListNode n = NULL;
  if (iter->current == iter->list->head) {
    printf("iter->current == iter->list->head\n");
    n = iter->list->head;
    if (iter->list->head == iter->list->last)
      iter->list->last = NULL;
    iter->list->head = iter->list->head->next;
    iter->current = iter->list->head;
  }
  else {
    printf("iter->current != iter->list->head\n");
    if (iter->previous == NULL)
      listError("listIteratorRemove: previous is NULL");
```

```
    n = iter->current;
    if (iter->current->next == NULL)
      iter->list->last = iter->previous;
    iter->previous->next = iter->current->next;
    iter->current = iter->previous;
  }
  iter->previous = NULL;
  disposeListNode(n);
  n = NULL;
  return iter->list;
}

/* disposeListIterator: deallocates memory occupied by ListIterator */
ListIterator disposeListIterator(ListIterator *iter)
{
  free(*iter);
  *iter = NULL;
  return *iter;
}

/* disposeList: deallocates memory occupied by list */
List disposeList(List *list)
{
  if (*list == NULL)
    listError("disposeList: list is NULL");

  ListNode p = (*list)->head;
  ListNode anchor = NULL;
  while (p != NULL) {
    anchor = p->next;
    disposeListNode(p);
    p = anchor;
  }
  free(*list);
  *list = NULL;
  return *list;
}
```

/* TestList.c */

```
#include <stdio.h>
#include "list.h"

int main()
{
  List list = newList();
  char chA = 'A';
  char chB = 'B';
  char chC = 'C';
  printf("Using addFirst(). Expect to see C, B, A: ");
  addFirst(&chA, list);
  addFirst(&chB, list);
  addFirst(&chC, list);
  ListIterator it = newListIterator(list);
  while (listIteratorHasNext(it)) {
    char ch = *(char *)listIteratorNext(it);
    printf("%c ", ch);
  }
  printf("\n\n");

  list = newList();
  printf("Using addLast(). Expect to see A, B, C: ");
  addLast(&chA, list);
  addLast(&chB, list);
  addLast(&chC, list);
  it = newListIterator(list);
  while (listIteratorHasNext(it)) {
    char ch = *(char *)listIteratorNext(it);
    printf("%c ", ch);
  }
```

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

  printf("Using getFirst(). Expect to see A: %c\n\n",
              *(char *)getFirst(list));
  printf("Using getLast(). Expect to see C: %c\n\n",
              *(char *)getLast(list));

  printf("Using removeFirst(), removeLast(). Expect to see B: ");
  removeFirst(list);
  removeLast(list);
  it = newListIterator(list);
  while (listIteratorHasNext(it)) {
    char ch = *(char *)listIteratorNext(it);
    printf("%c ", ch);
  }
  printf("\n\n");

  printf("Using listSize().  Expect to see 1, 2, 3, 0: ");
  printf("%d ", listSize(list));
  addFirst(&chA, list);
  printf("%d ", listSize(list));
  addLast(&chC, list);
  printf("%d ", listSize(list));
  list = newList();
  printf("%d ", listSize(list));
  printf("\n\n");

  printf("Using listIteratorHasPrevious() and
              listIteratorPrevious()\n");
  printf("Expect to see C, B, A: ");
  addLast(&chA, list);
  addLast(&chB, list);
  addLast(&chC, list);
  it = newListIterator(list);
  while (listIteratorHasPrevious(it)) {
    char ch = *(char *)listIteratorPrevious(it);
    printf("%c ", ch);
  }
  printf("\n\n");

  return 0;
}
```



5.

```
/* queue.c */

#include "queue.h"
#include "list.h"

/* queueError: reports queue errors, halts program execution */
```

```
int queueError(char *error)
{
  return listError(error);
}

/* newQueue: returns a new, empty queue */
Queue newQueue()
{
  return (Queue)newList();
}

/* isEmptyQueue: returns 0 (true) if queue has no elements */
int isEmptyQueue(const Queue q)
{
  return isEmptyList((List)q);
}

/* joinQueue: adds data item to end of queue */
Queue joinQueue(void *data, Queue q)
{
  return (Queue)addLast(data, (List)q);
}

/* headOfQueue: returns item at head of queue */
void *headOfQueue(const Queue q)
{
  return getFirst((List)q);
}

/* leaveQueue: removes data item from front of queue */
Queue leaveQueue(Queue q)
{
  return (Queue)removeFirst((List)q);
}

/* disposeQueue: deallocates memory occupied by queue */
Queue disposeQueue(Queue *q)
{
  return (Queue)disposeList((List *)q);
}
```

Exercise 6.1

1. The cardinality of a set is the number of elements it contains.

```
/* cardinality: returns the number of elements in the given set */
int cardinality(const Set s)
{
  assert(s != NULL);
  int i = 0;

  SetIterator it = newSetIterator(s);
  while (setIteratorHasNext(it)) {
    setIteratorNext(it);
    i++;
  }
  return i;
}
```

2. Set *s* is a subset of set *t* if every element in *s* is also in *t*.

```
/* isASubset: returns 1 if s is a subset of t */
int isASubset(const Set s, const Set t)
{
  SetIterator it = newSetIterator(s);
  while (setIteratorHasNext(it)) {
    if (!setContains(t, setIteratorNext(it)))
      return 0;
  }
  return 1;
}
```

3. Two sets are equal if they have identical elements, i.e. if they are both subsets of each other.

```
/* equalSets: returns 1 if the two given have identical elements */
int equalSets(const Set s, const Set t)
{
  return isASubset(s, t) && isASubset(t, s);
}
```