Computer Programming

Dynamic Data Structures

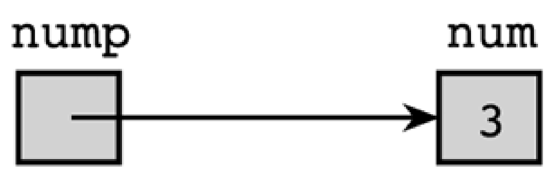
Introduction

- Dynamic data structures: expands and contracts as the program executes
  - Decision on space is made during execution
  - Array: decision is made beforehand
    - Partially filled array is possible but still maximum size is decide before compilation
  - Required dynamic memory allocation
    - Allocate space as necessary during execution
- Linked list: linear sequence of nodes
  - Nodes: a structure that points another structure
  - Can be used to form lists, stacks, queues

Pointers

- Used extensively for dynamic data structures
- Pointer Review:
  - Reference / indirect access

Comparison of Pointer and Nonpointer Variables



Reference	Explanation	Value
num	Direct value of num	3
nump	Direct value of nump	Pointer to location containing 3
*nump	Indirect value of nump	3

Pointers

- Used extensively for dynamic data structures
- Pointer Review:
  - Reference / indirect access
  - Function parameters
    - Output parameter (Ex: long division)
    - Input parameter

Pointers as Output Parameters

```
#include <stdio.h>

void long_division(int dividend, int divisor, int *quotientp,
                   int *remainderp);

int
main(void)
{
   int quot, rem;

   long_division(40, 3, &quot, &rem);
   printf("40 divided by 3 yields quotient %d ", quot);
   printf("and remainder %d\n", rem);
   return (0);
}

/*
 *  Performs long division of two integers, storing quotient
 *  in variable pointed to by quotientp and remainder in
 *  variable pointed to by remainderp
 */
void long_division(int dividend, int divisor, int *quotientp,
                   int *remainderp)
{
   *quotientp = dividend / divisor;
   *remainderp = dividend % divisor;
}
```

Pointers

- Used extensively for dynamic data structures
- Pointer Review:
  - Reference / indirect access
  - Function parameters
    - Output parameter (Ex: long division)
    - Input parameter
  - Representing arrays and strings
    - Passing as a parameter
  - Pointers to structures
  - File pointers

Pointers

- Used extensively for dynamic data structures
- Pointer Review:
  - Reference / indirect access
  - Function parameters
  - Representing arrays and strings
  - Pointers to structures
- Operations with pointers
  - Indirection
  - Assignment
  - Equality operators (== , !=)
  - Increment, decrement

Dynamic Memory Allocation

- Pointer declaration does not allocate memory for values.

```
double * nump;
```

- Use function malloc to allocate memory

```
malloc(sizeof(double))
```

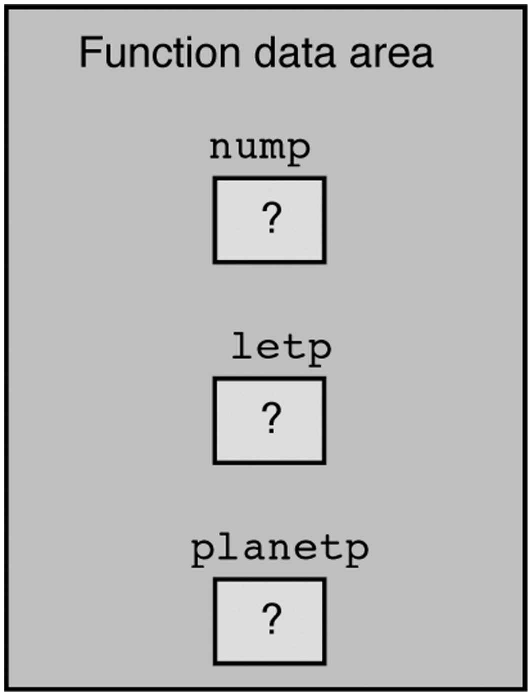
  - Allocates number of bytes defined by the parameter
  - Memory allocated in the heap (not stack)
  - Returns a pointer to the block allocated
  - Memory allocated by malloc could be used to store any value
  - What should be the return type? (type of the pointer)

```
void *
```

```
nump = (double*)malloc(sizeof(double));
```

Three Pointer-Type Local Variables

```
int       *nump;
char      *letp;
planet_t *planetp;
```

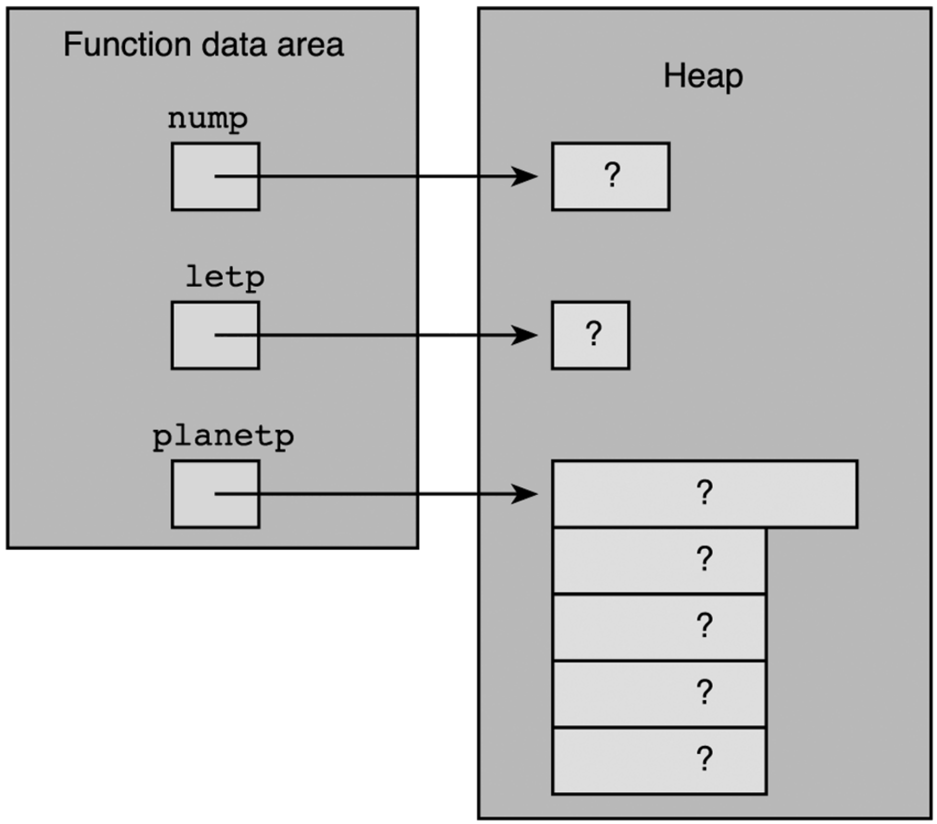


Dynamic Allocation of Variables

nump = (int *)malloc(sizeof(int));

letp = (char *)malloc(sizeof(char));

planetp = (planet_t *)malloc(sizeof(planet_t));

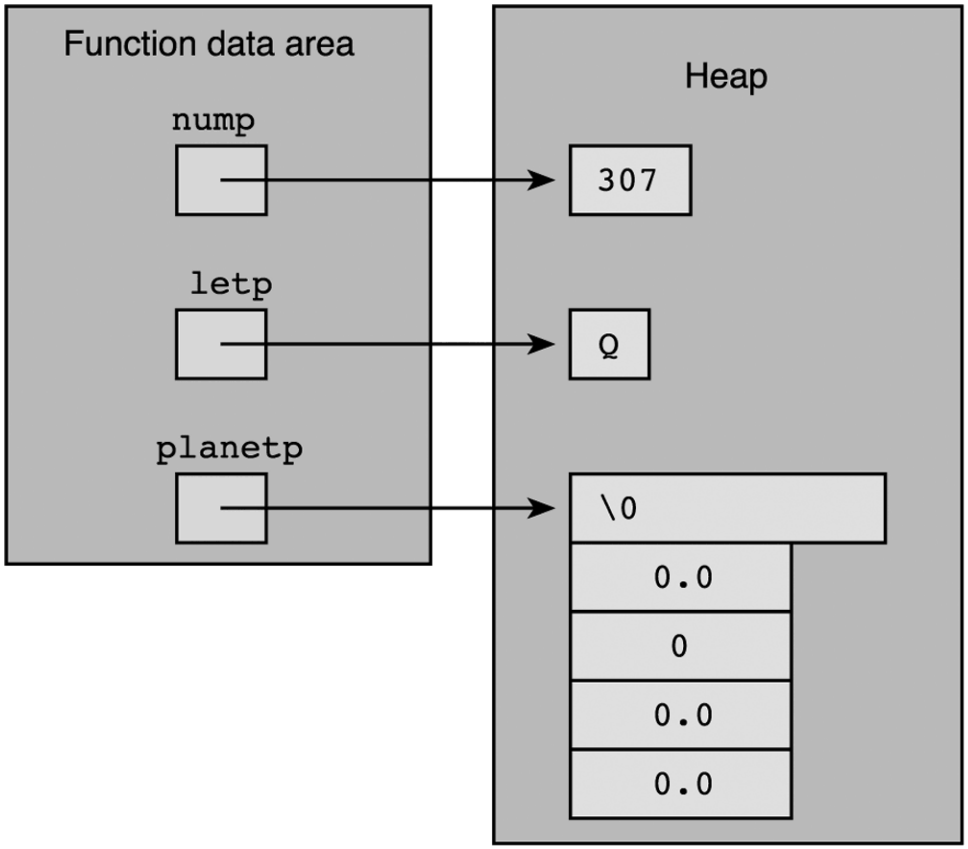


Assignments to Dynamically Allocated Variables

*nump = 307;

*letp = ‘Q’;

*planetp = blank_planet;



Components of Dynamically Allocated Structure

- Indirection + component selection

  (*planetp).name

- Indirect component selection

  planetp->name

```
printf("%s\n", planetp->name);
printf("  Equatorial diameter:  %.0f km\n", planetp->diameter);
printf("  Number of moons:  %d\n", planetp->moons);
printf("  Time to complete one orbit of the sun:  %.2f years\n",
       planetp->orbit_time);
printf("  Time to complete one rotation on axis:  %.4f hours\n",
       planetp->rotation_time);
```

Allocation of Arrays with calloc

- malloc : to allocate single memory block
- calloc : to dynamically create array of elements
  - Elements of any type (built-in or user defined)
- calloc : two arguments
  - Number of elements
  - Size of one element
- Allocates the memory and initializes to zero
- Returns a pointer

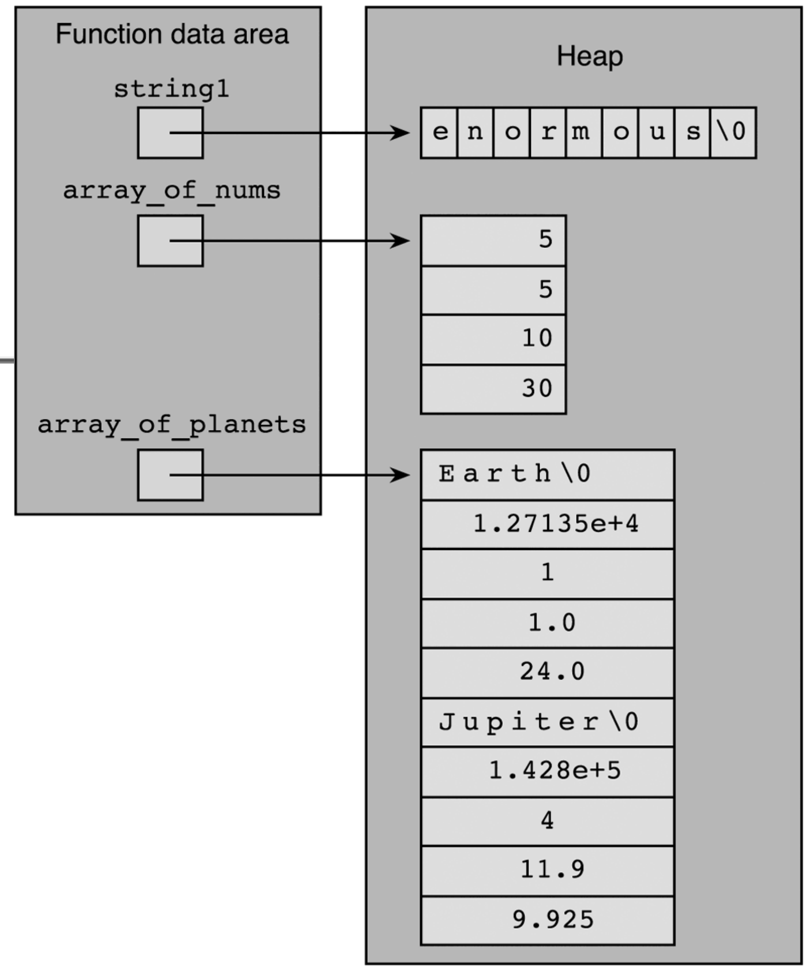
Allocation of Arrays with calloc

```
#include <stdlib.h>  /* gives access to calloc */
int scan_planet(planet_t *plnp);

int
main(void)
{
      char     *string1;
      int      *array_of_nums;
      planet_t *array_of_planets;
      int       str_siz, num_nums, num_planets, i;
      printf("Enter string length and string> ");
      scanf("%d", &str_siz);
      string1 = (char *)calloc(str_siz, sizeof (char));
      scanf("%s", string1);

      printf("\nHow many numbers?> ");
      scanf("%d", &num_nums);
      array_of_nums = (int *)calloc(num_nums, sizeof (int));
      array_of_nums[0] = 5;
      for (i = 1;  i < num_nums;  ++i)
            array_of_nums[i] = array_of_nums[i - 1] * i;

      printf("\nEnter number of planets and planet data> ");
      scanf("%d", &num_planets);
      array_of_planets = (planet_t *)calloc(num_planets,
                                            sizeof (planet_t));
```

(continued)

Allocation of Arrays with calloc

```
    for (i = 0;  i < num_planets;  ++i)
         scan_planet(&array_of_planets[i]);
    . . .
}

Enter string length and string> 9 enormous

How many numbers?> 4

Enter number of planets and planet data> 2
Earth    12713.5 1  1.0 24.0
Jupiter 142800.0 4 11.9  9.925
```



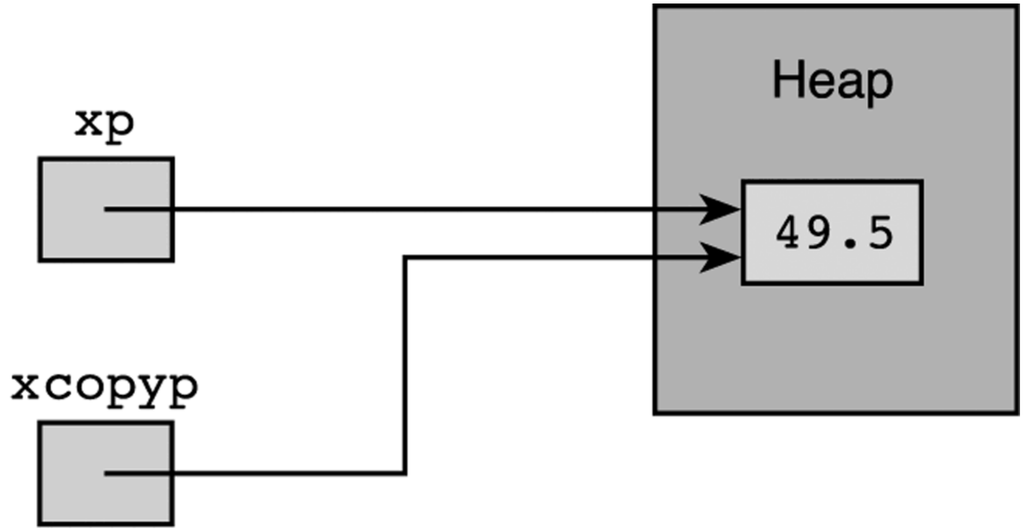
Returning Memory

- free : returns memory cells to heap
  - Allocated by calloc or malloc
  - Returned memory can be allocated later

```
free(nump);
free(array_of_planets);
```

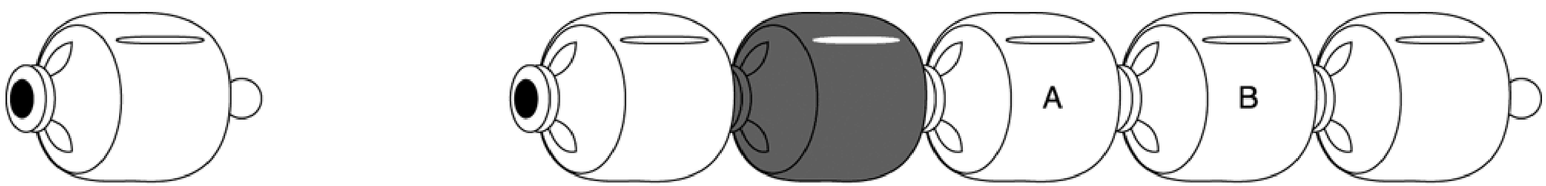
Multiple Pointers to a Cell

```
double *xp, *xcopyp;
xp = (double *)malloc(sizeof(double));
*xp = 49.5;
xcopyp = xp;
free(xp);
*xcopyp = 52.25;
```



Linked Lists

- A sequence of nodes
  - Each node is connected to the following one
  - Like pop beads
    - A chain can be formed easily
    - Modified easily (remove and insert)



Pop bead

Chain of pop beads

Node Structure

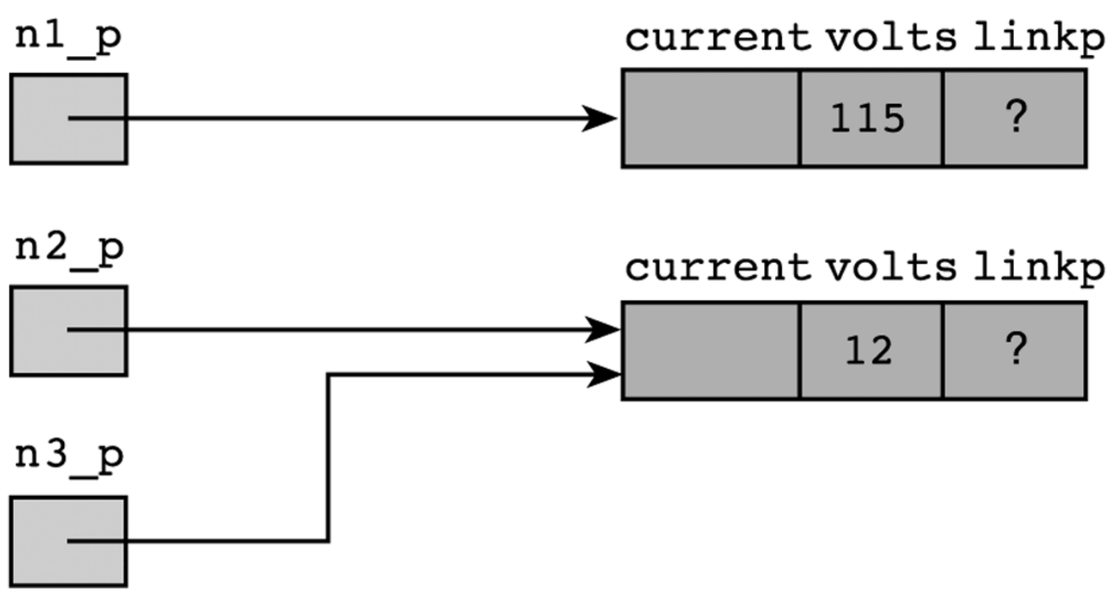
- Structure with pointer components
- Allocate a node as necessary
  - And connect then to form a linked list

```
typedef struct node_s {
        char            current[3];
        int             volts;
        struct node_s *linkp;
} node_t;
```

  - Use a structure tag

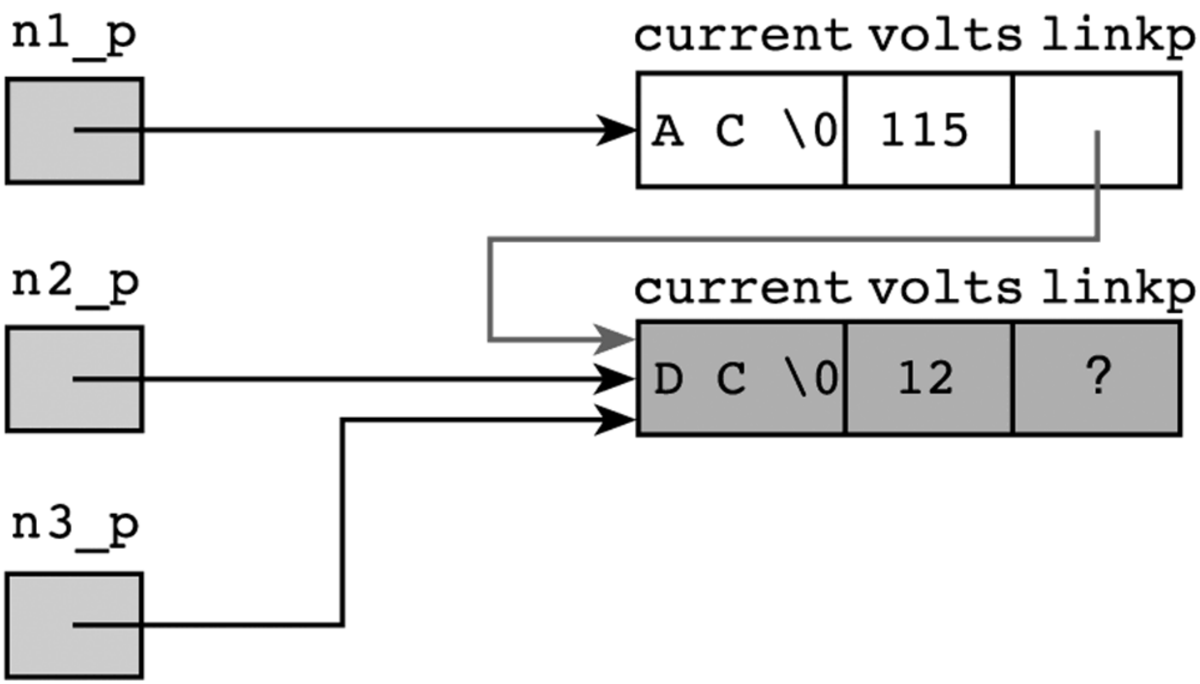
Multiple Pointers to the Same Structure

```
node_t *n1_p, *n2_p, *n3_p;
n1_p = (node_t *)malloc(sizeof(node_t));
n1_p->volts = 115;
n2_p = (node_t *)malloc(sizeof(node_t));
n2_p->volts = 12;
n3_p = n2_p;
```



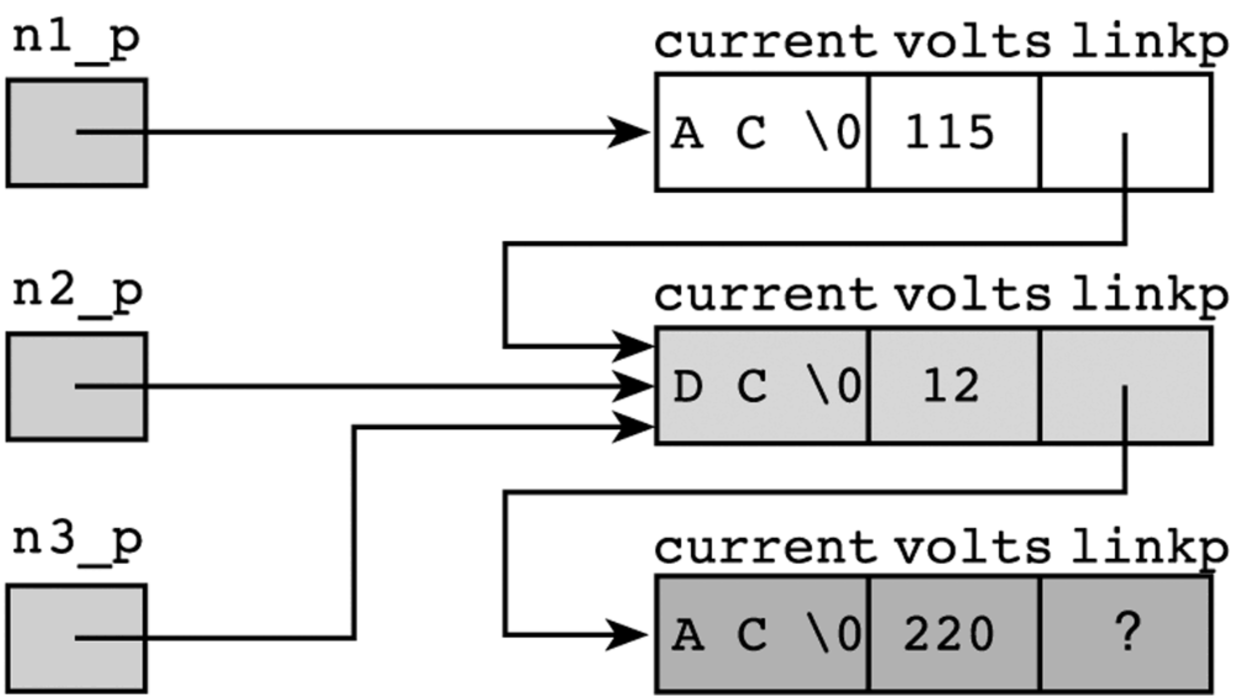
Linking Two Nodes

n1_p->linkp = n2_p;



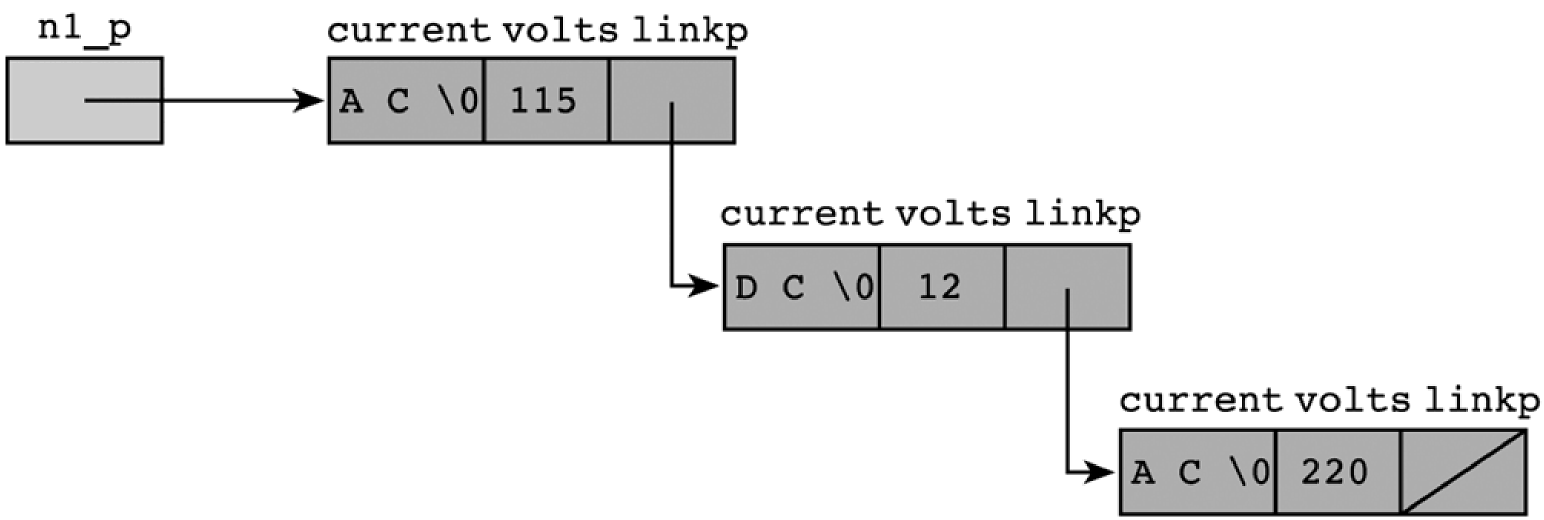
Three-Node Linked List

```
n2_p->linkp = (node_t *)malloc(sizeof(node)t));
n2_p->linkp->volts = 220;
strcpy(n2_p->linkp->current, "AC");
```



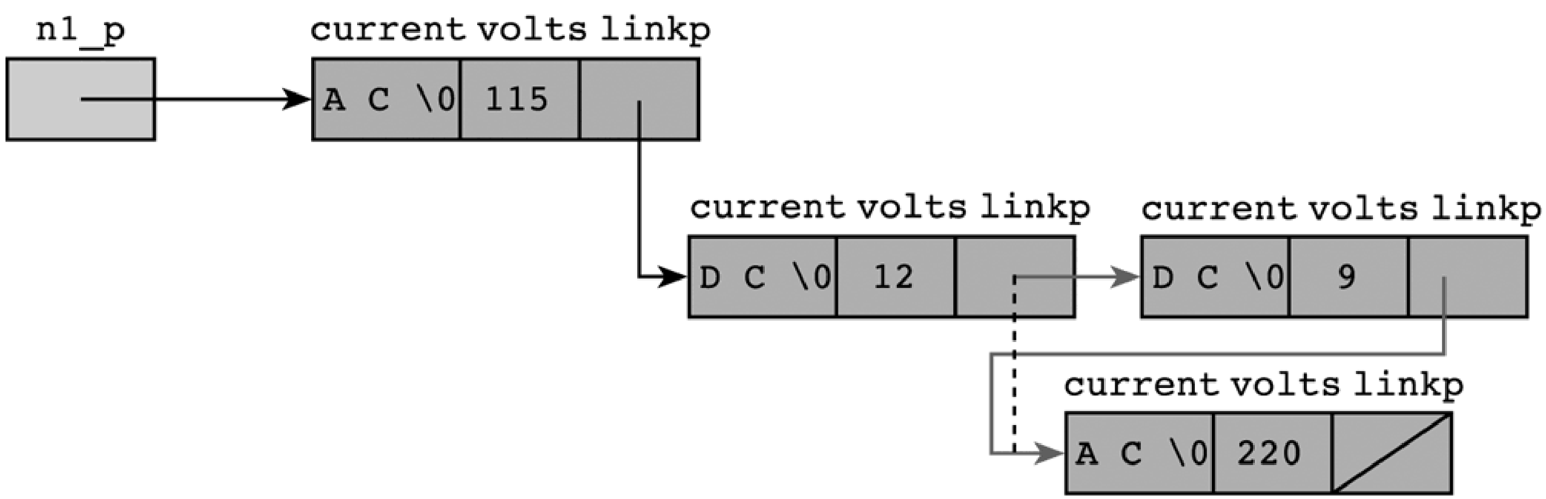
Three-Element Linked List

- List head
- End of list indicator
- Empty list



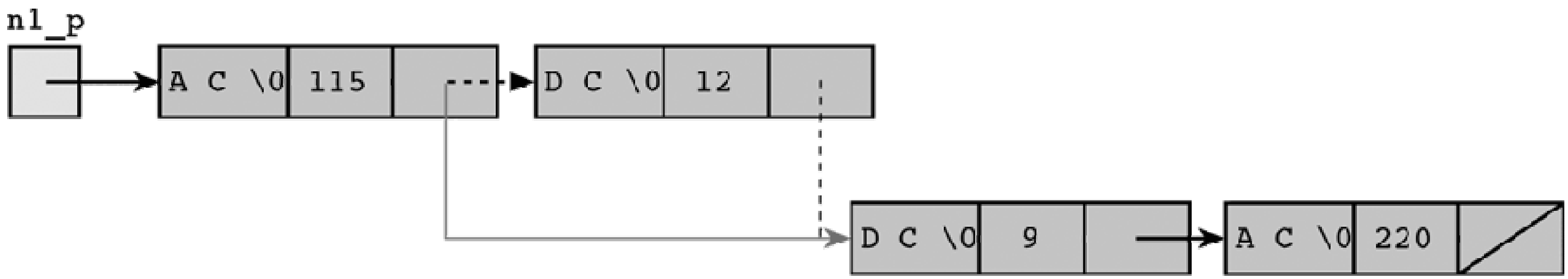
Linked List After an Insertion

- Linked List can be modified easily
  - Insertion



Linked List After a Deletion

- Linked List can be modified easily
  - Insertion
  - Deletion



Linked List Operations

- Assume following declaration

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

- Traversing a list
  - Process each node in sequence
  - Start with the list head and follow list pointers
- Operations should take list head as a parameter

Recursive function print_list

```
/*
 *  Displays the list pointed to by headp
 */
void
print_list(list_node_t *headp)
{
```

Recursive function print_list

```
/*
 *  Displays the list pointed to by headp
 */
void
print_list(list_node_t *headp)
{
      if (headp == NULL) {    /* simple case - an empty list                  */
            printf("\n");
      } else {                /* recursive step - handles first element       */
            printf("%d", headp->digit);  /*      leaves rest to                */
            print_list(headp->restp);    /*      recursion                     */
      }
}
```

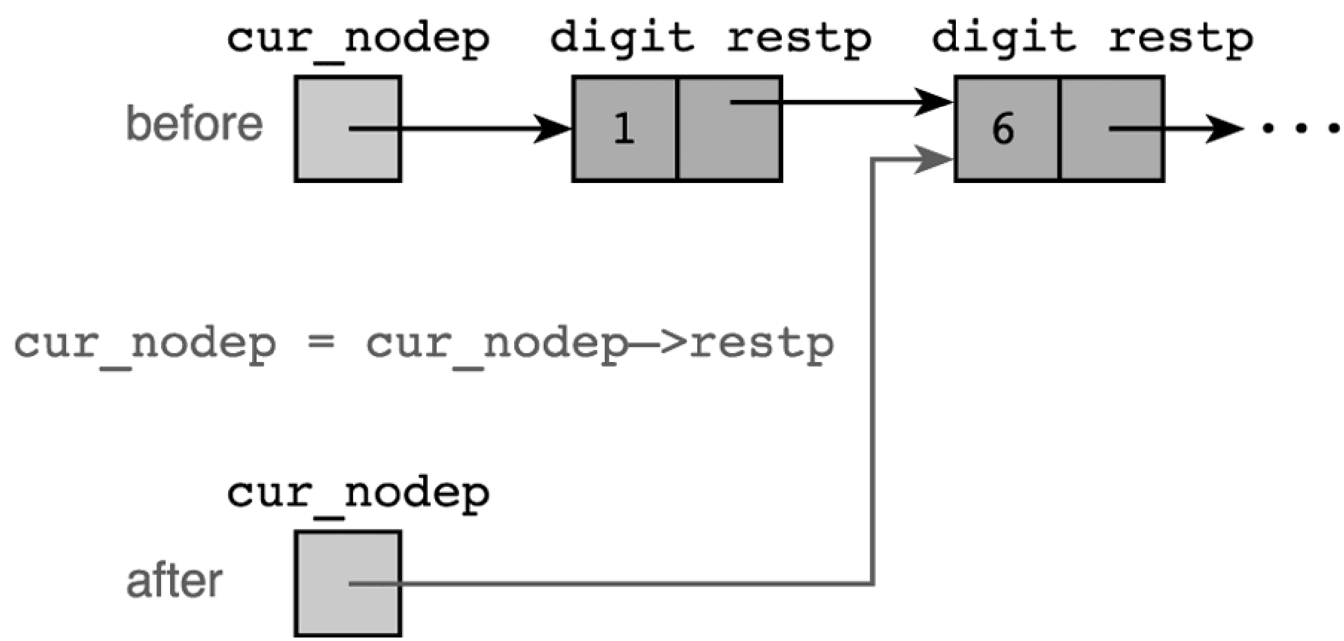
Recursive and Iterative List Printing

- Tail recursion: recursive call is at the last step
  - Easy to convert it to recursion

```
/* Displays the list pointed to by headp */
void
print_list(list_node_t *headp)
{
 if (headp == NULL) {/* simple case */
     printf("\n");
 } else {           /* recursive step */
     printf("%d", headp->digit);
     print_list(headp->restp);
 }
}
```

```
{ list_node_t *cur_nodep;

  for  (cur_nodep = headp; /* start at
                              beginning  */
        cur_nodep != NULL; /* not at
                              end yet    */
        cur_nodep = cur_nodep->restp)
      printf("%d", cur_nodep->digit);
  printf("\n");
}
```

Update of List-Traversing Control Variable



Recursive Function get_list

```
#include <stdlib.h>  /*  gives access to malloc */
#define SENT -1
/*
 *  Forms a linked list of an input list of integers
 *  terminated by SENT
 */
list_node_t *
get_list(void)
{
```

Recursive Function get_list

```
#include <stdlib.h>  /*  gives access to malloc */
#define SENT -1
/*
 *  Forms a linked list of an input list of integers
 *  terminated by SENT
 */
list_node_t *
get_list(void)
{
      int data;
      list_node_t *ansp;

      scanf("%d", &data);
      if (data == SENT) {
            ansp = NULL;
      } else {
            ansp = (list_node_t *)malloc(sizeof (list_node_t));
            ansp->digit = data;
            ansp->restp = get_list();
      }

      return (ansp);
}
```

```c
/*
 *  Forms a linked list of an input list of integers terminated by SENT
 */
list_node_t *
get_list(void)
{
      int data;
      list_node_t *ansp,
                  *to_fillp, /* pointer to last node in list whose
                                restp component is unfilled          */
                  *newp;     /* pointer to newly allocated node     */

      /* Builds first node, if there is one */
      scanf("%d", &data);
      if (data == SENT) {
            ansp = NULL;
      } else {
            ansp = (list_node_t *)malloc(sizeof (list_node_t));
            ansp->digit = data;
            to_fillp = ansp;

            /* Continues building list by creating a node on each
               iteration and storing its pointer in the restp component of the
               node accessed through to_fillp */
            for  (scanf("%d", &data);
                  data != SENT;
                  scanf("%d", &data)) {
                newp = (list_node_t *)malloc(sizeof (list_node_t));
                newp->digit = data;
                to_fillp->restp = newp;
                to_fillp = newp;
            }

            /* Stores NULL in final node's restp component */
            to_fillp->restp = NULL;
      }
      return (ansp);
}
```

Function search

```
/*
 *  Searches a list for a specified target value.  Returns a pointer to
 *  the first node containing target if found.  Otherwise returns NULL.
 */
list_node_t *
search(list_node_t *headp,   /* input - pointer to head of list  */
       int          target)   /* input - value to search for     */
{
```

Function search

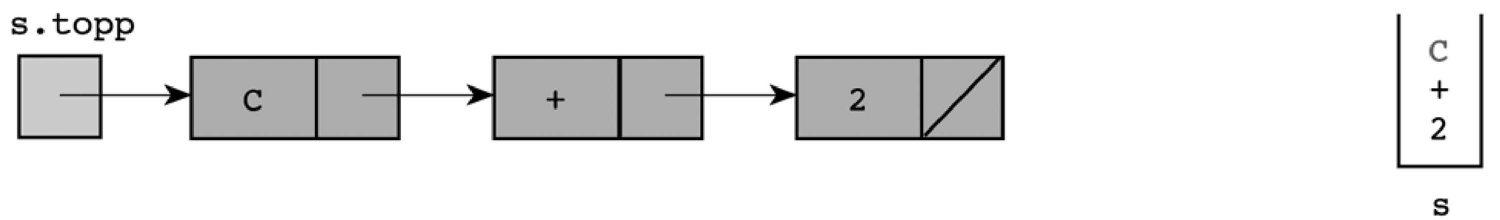
```
/*
 *  Searches a list for a specified target value.  Returns a pointer to
 *  the first node containing target if found.  Otherwise returns NULL.
 */
list_node_t *
search(list_node_t *headp,   /* input - pointer to head of list  */
       int          target)   /* input - value to search for     */
{
     list_node_t *cur_nodep;  /* pointer to node currently being checked */

     for  (cur_nodep = headp;
           cur_nodep != NULL  &&  cur_nodep->digit != target;
           cur_nodep = cur_nodep->restp) {}

      return (cur_nodep);
}
```

Stack

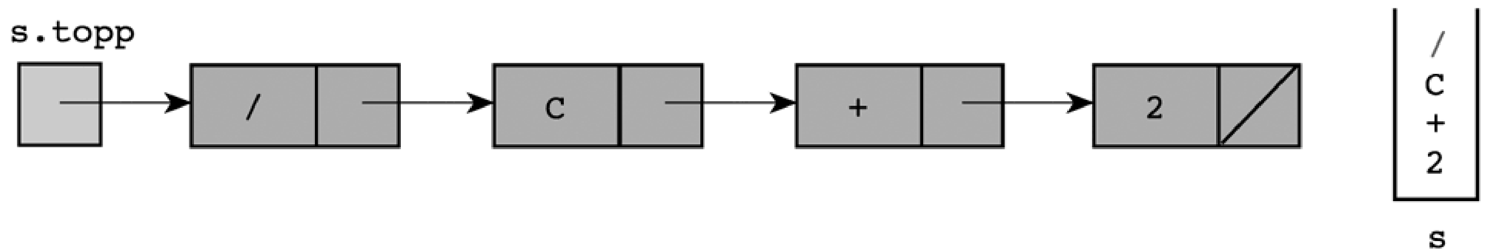
Linked List Representation of Stacks

- Stack can be implemented using array
- Stack = LIFO List
  - Linked list can be used

Stack of three characters



Stack after insertion (push) of '/'



Structure Types

```
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;
```

Stack Manipulation with 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)

Stack Manipulation with push and pop

```c
#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);
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);
}
```

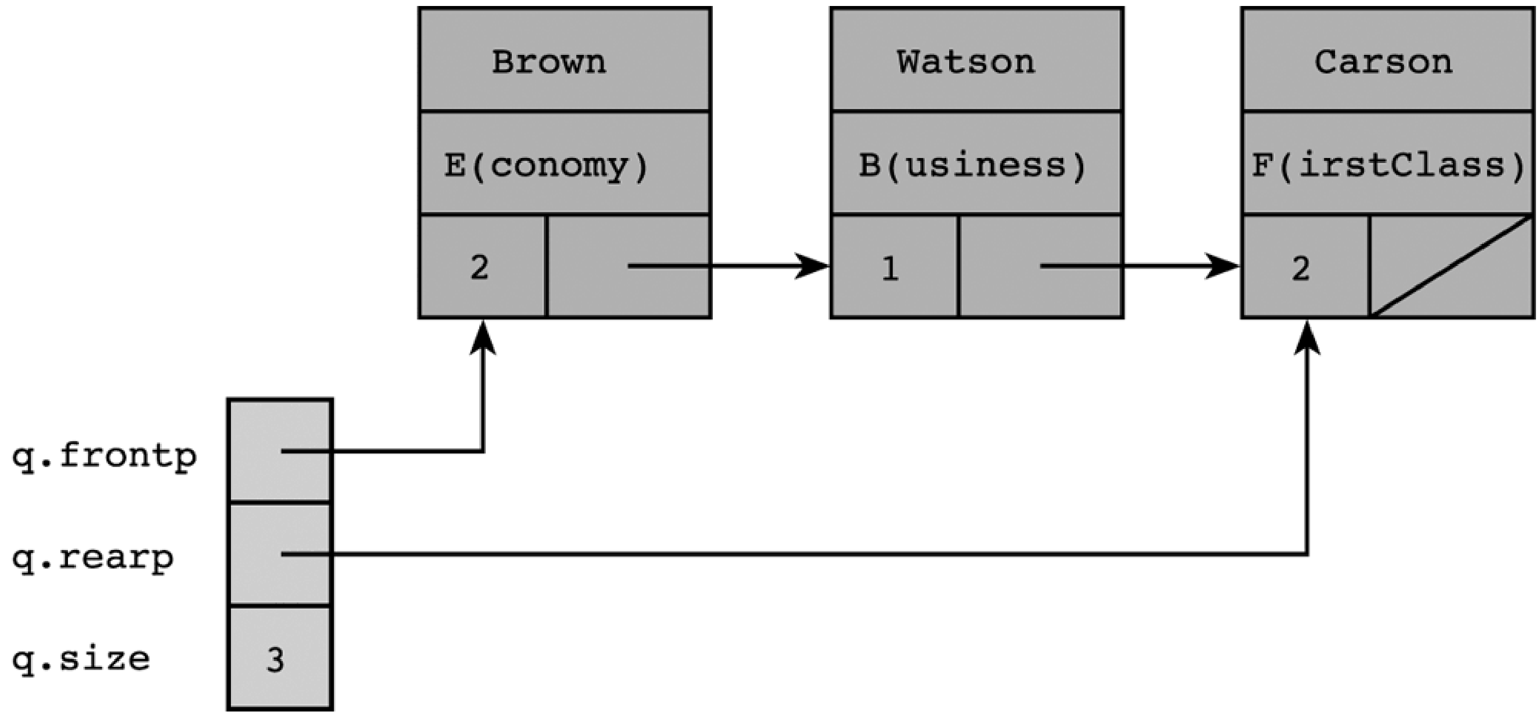
```
/*
 *  The value in c is placed on top of the stack accessed through sp
 *  Pre:  the stack is defined
 */
void
push(stack_t         *sp, /* input/output - stack              */
     stack_element_t c)   /* input         - element to add */
{
      stack_node_t *newp;  /* pointer to new stack node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

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

Queue

- Queue = FIFO List
- EX: Model a line of customers waiting at a checkout counter
- Need to keep both ends of a queue
  - Front and rear

A Queue of Passengers



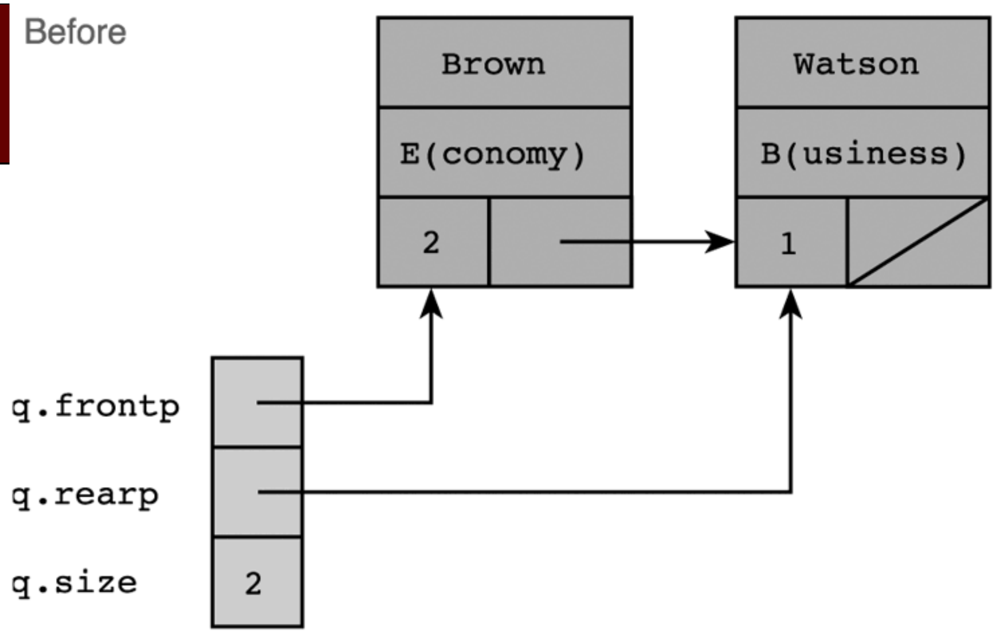
Structure Types

```
/*  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;
```

```c
/*
 *  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;

```

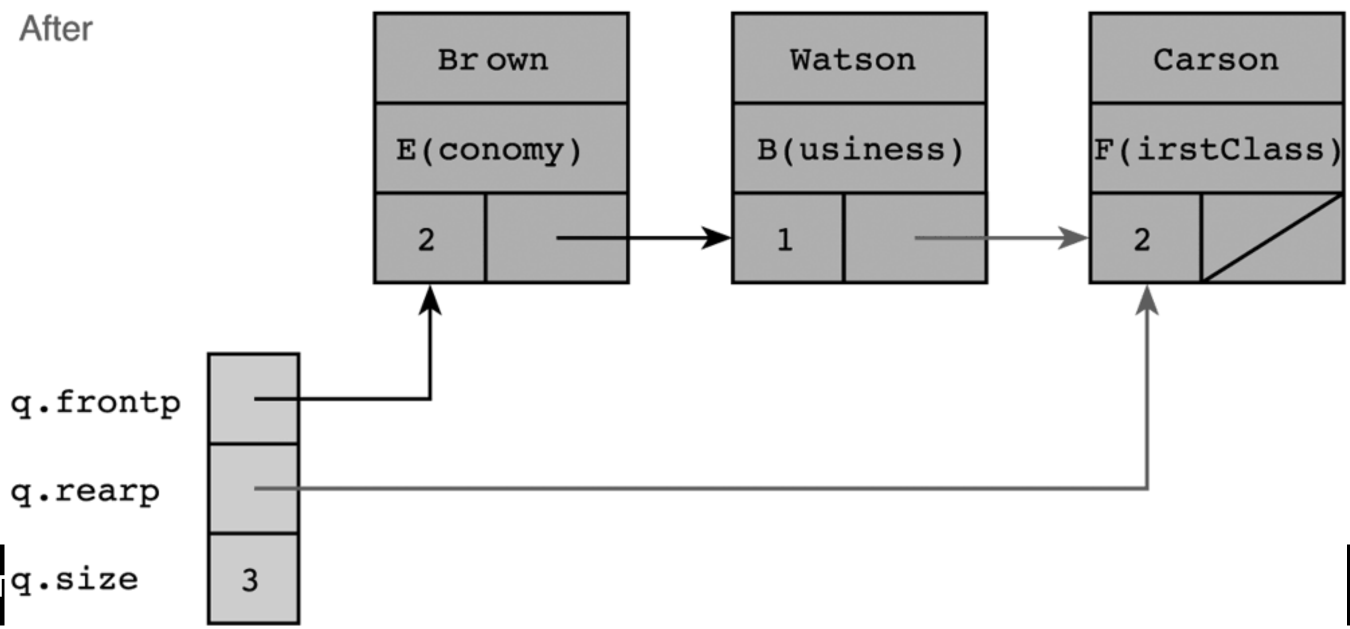
```
        case 'R':
            if (pass_q.size > 0) {
                fst_pass = remove_from_q(&pass_q);
                printf("Passenger removed from queue: \n");
                print_passenger(fst_pass);
            } else {
                printf("Queue empty - noone to delete\n");
         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

Before



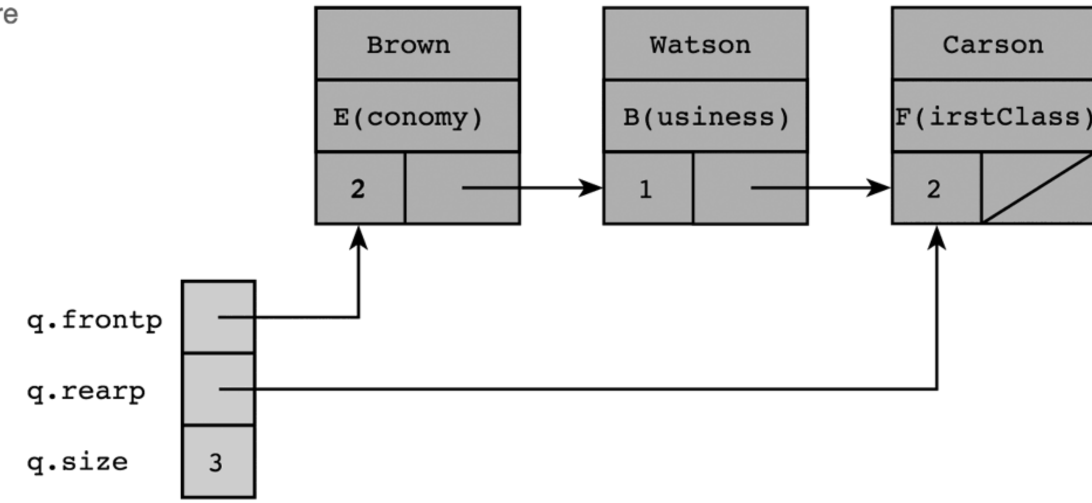

```
add_to_q(&q, next_pass);
```

After



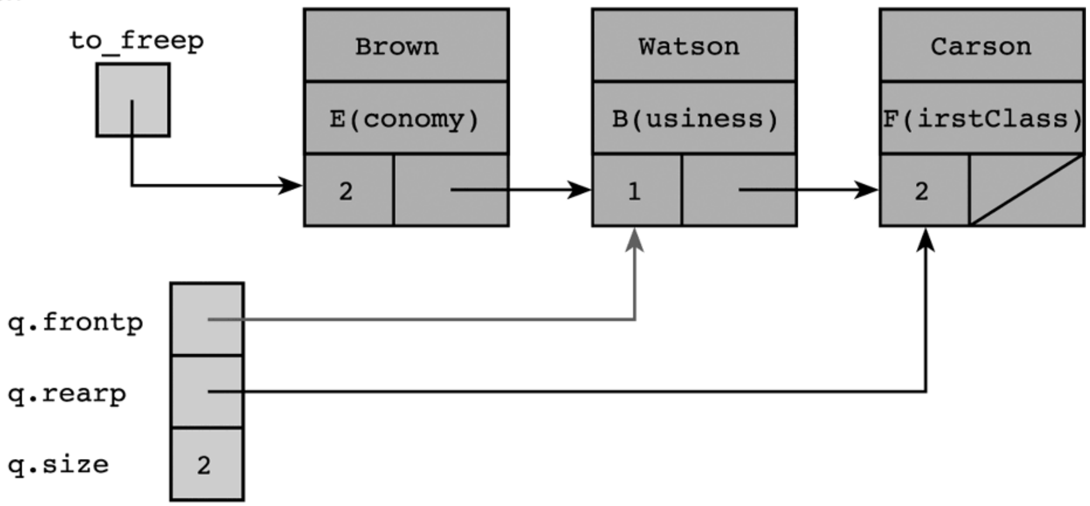
```c
/*
 *  Adds ele at the end of queue accessed through qp
 *  Pre:  queue is not empty
 */
void
add_to_q(queue_t          *qp,  /* input/output - queue   */
         queue_element_t  ele) /* input - element to add */
{
      if (qp->size == 0) {                  /* adds to empty queue              */
            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);
}
```

Before

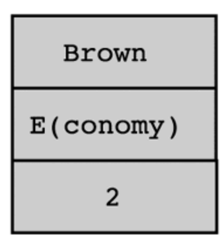



```
remove_fr om_q(&q);
```

During function cal l



Value returned by `remove_from_q`



```c
/*
 *  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                               */
      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);
}
```

Case Study: Ordered Lists

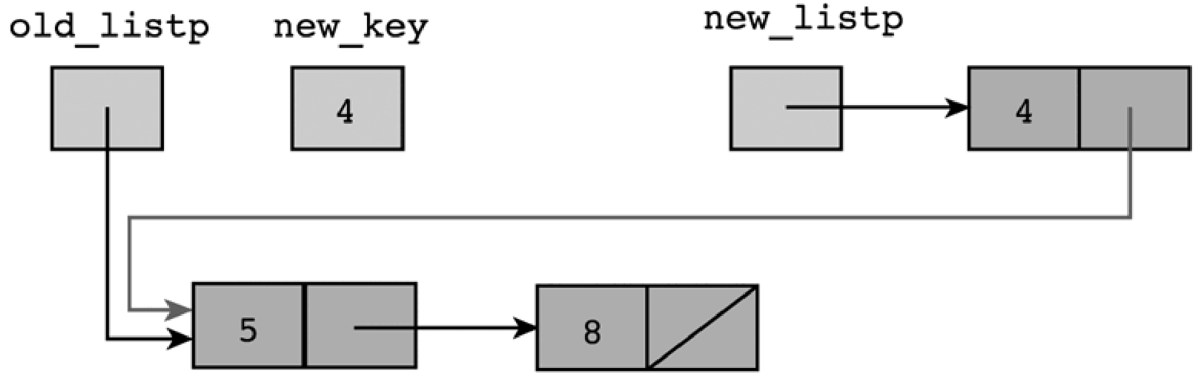
- The position of elements is determined by their key value
  - Increasing or decreasing order

```
/*
 *  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;

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
```

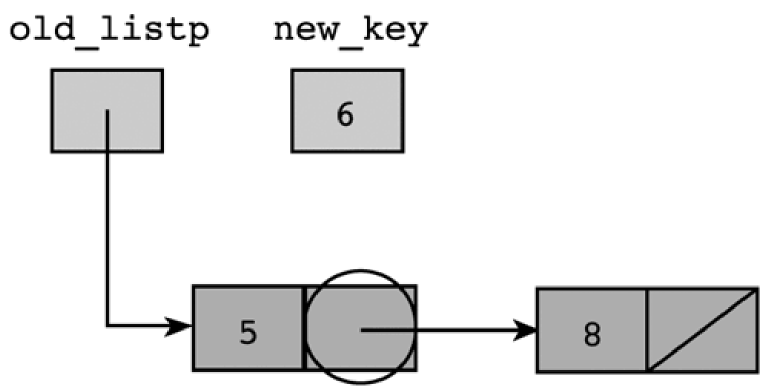
```
int
main(void)
{
      int            next_key;
      ordered_list_t my_list = {NULL, 0};

      /* 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);
          }
      }

      return (0);
}
```

Simple Case 1

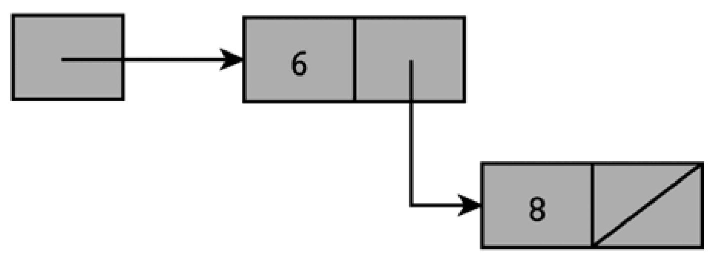
Simple Case 2



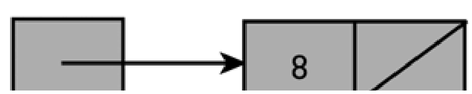
Recursive Step



`new_listp` is `old_listp` with circled component changed to



which is the result of inserting 6 in order in



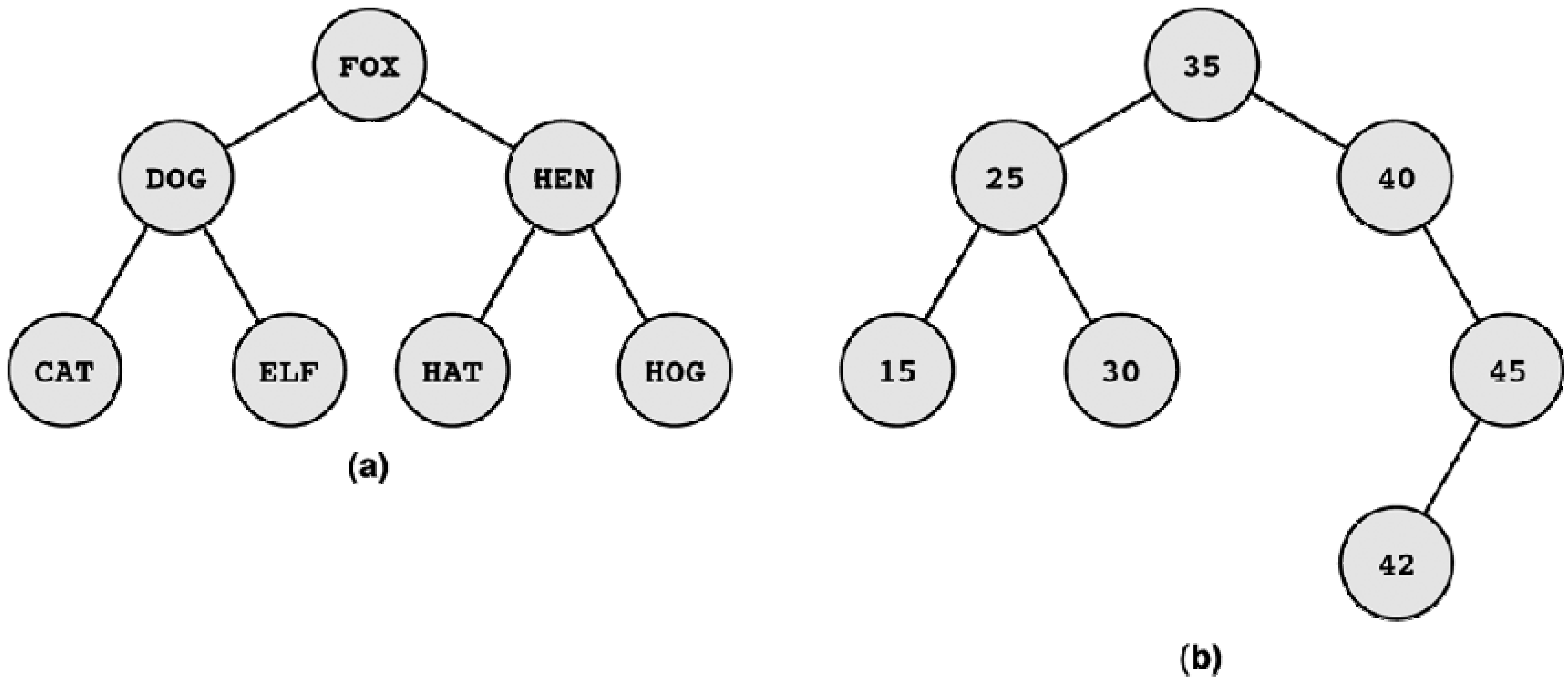
```c
/*
 *  Inserts a new node containing new_key in order in old_list, returning as
 *  the function value a pointer to the first node of the new list
 */
list_node_t *
insert_in_order(list_node_t *old_listp,   /* input/output */
                int          new_key)     /* input        */
{
      list_node_t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

```c
/*
 *  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);
}
```

```
/*
 *  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);
}
```

```
/*
 *  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->restp;
            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->restp = delete_ordered_node(listp->restp, target,
                                              is_deletedp);
      }

      return (ansp);
}
```

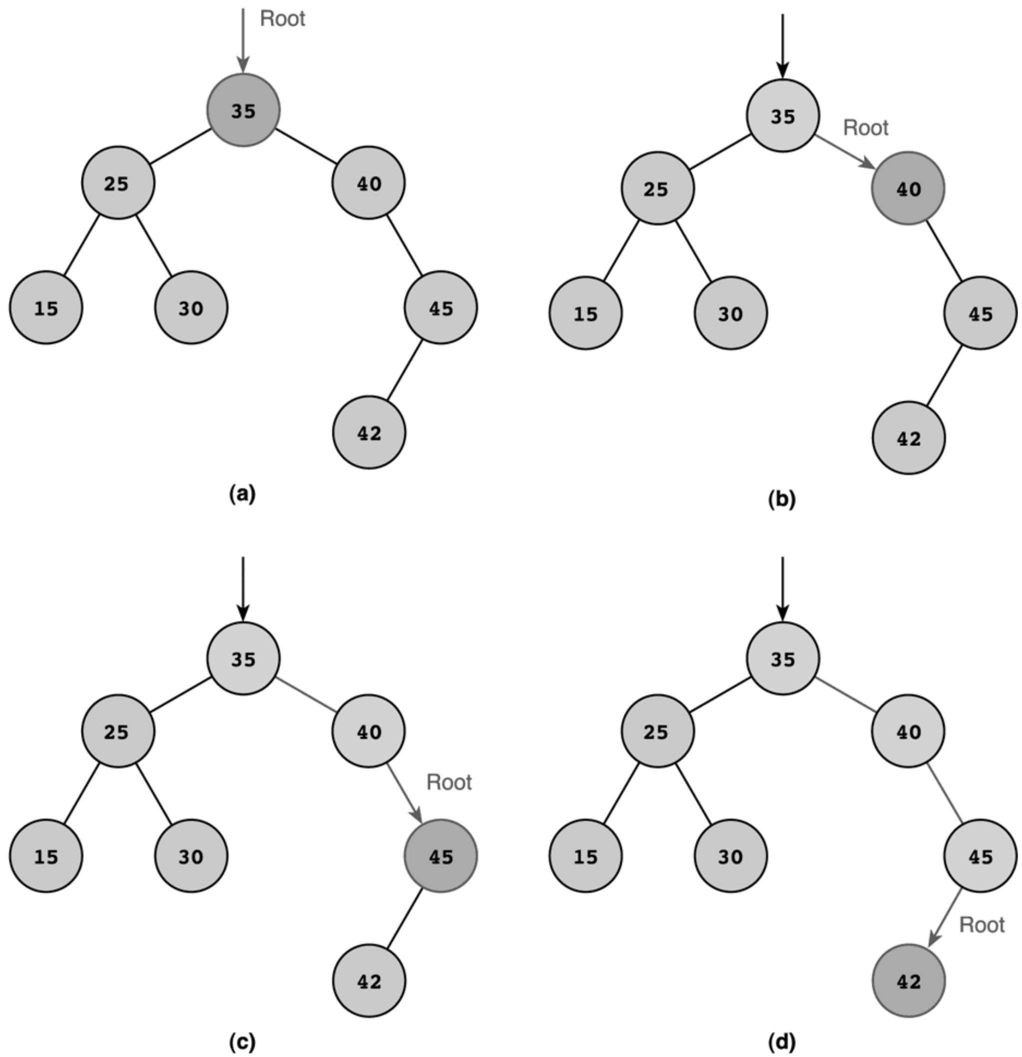
Binary Trees



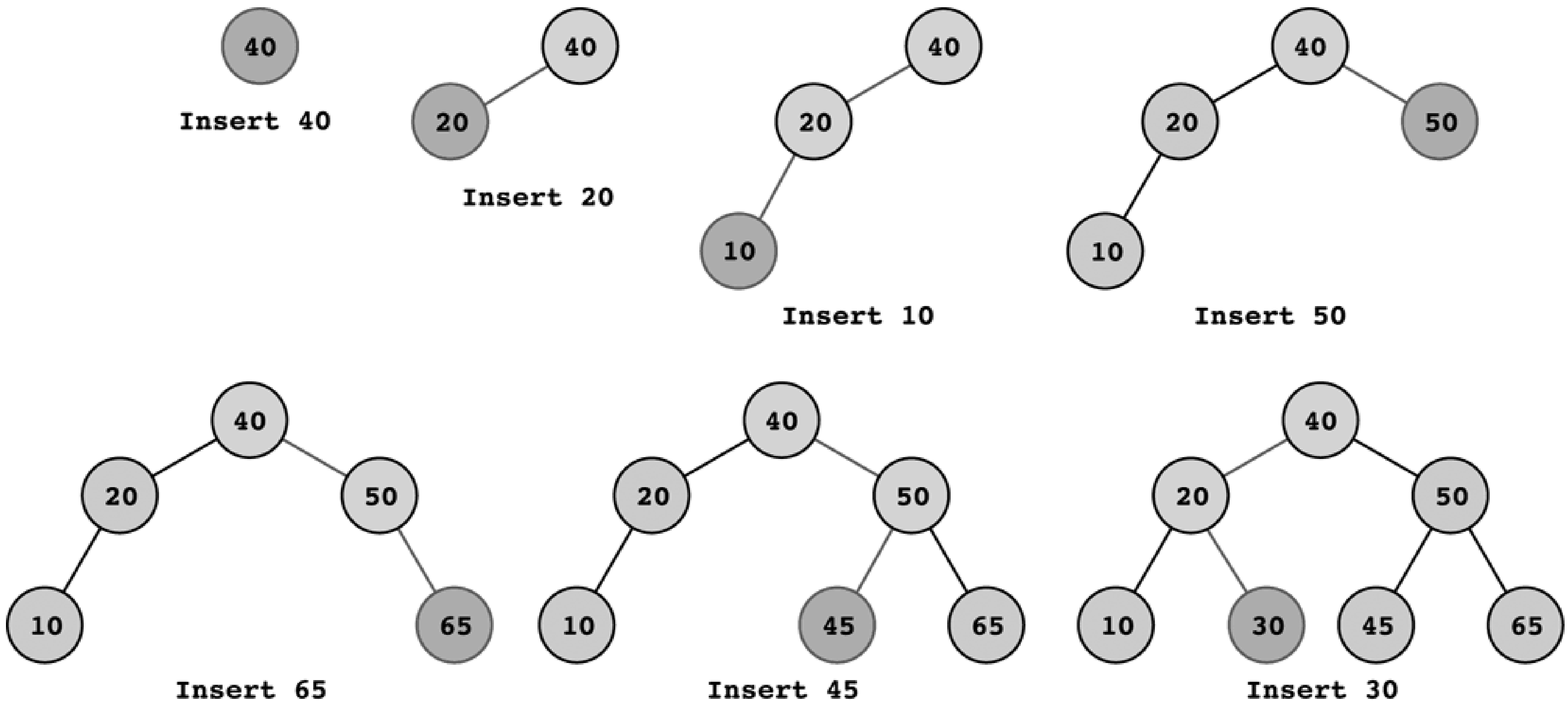
Binary Search Trees

- How to implement?

Binary Tree Search for 42



Building a Binary Search Tree



Creating a Binary Search Tree

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

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

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

typedef struct tree_node_s {
      int                   key;
      struct tree_node_s *leftp, *rightp;
} tree_node_t;

tree_node_t *tree_insert(tree_node_t *rootp, int new_key);
void tree_inorder(tree_node_t *rootp);
```

```
int
main(void)
{
      tree_node_t *bs_treep;   /* binary search tree              */
      int          data_key;   /* input - keys for tree           */
      int          status;     /* status of input operation       */

      bs_treep = NULL;   /* Initially, tree is empty */

      /* As long as valid data remains, scan and insert keys,
         displaying tree after each insertion. */
      for  (status = scanf("%d", &data_key);
            status == 1;
            status = scanf("%d", &data_key)) {
          bs_treep = tree_insert(bs_treep, data_key);
          printf("Tree after insertion of %d:\n", data_key);
          tree_inorder(bs_treep);
      }

      if (status == 0) {
            printf("Invalid data >>%c\n", getchar());
      } else {
            printf("Final binary search tree:\n");
            tree_inorder(bs_treep);
      }

      return (0);
}
```

(continued)

```
/*
 *  Insert a new key in a binary search tree.  If key is a duplicate,
 *  there is no insertion.
 *  Pre:  rootp points to the root node of a binary search tree
 *  Post: Tree returned includes new key and retains binary
 *        search tree properties.
 */
tree_node_t *
tree_insert(tree_node_t *rootp,   /* input/output - root node of
                                     binary search tree    */
            int          new_key) /* input - key to insert */
{
      if (rootp == NULL) {               /* Simple Case 1 - Empty tree   */
            rootp = TYPED_ALLOC(tree_node_t);
            rootp->key = new_key;
            rootp->leftp = NULL;
            rootp->rightp = NULL;
      } else if (new_key == rootp->key) {              /* Simple Case 2 */
            /* duplicate key - no insertion                             */
      } else if (new_key < rootp->key) {               /* Insert in     */
            rootp->leftp = tree_insert                 /* left subtree  */
                               (rootp->leftp, new_key);
      } else {                           /* Insert in right subtree */
            rootp->rightp = tree_insert(rootp->rightp,
                                        new_key);
      }

      return (rootp);
}
```