"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

- Alan Perlis

CSE102
Computer Programming with C

2020-2021 Spring Semester

Dynamic Data Structures

These slides are largely adapted from J.R. Hanly, E.B. Koffman, F.E. Sevilgen, and others…

1

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 to another structure
  - Can be used to form lists, stacks, queues

2

Pointers

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

3

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

4

Pointer and Nonpointer Variables

```c
#include <stdio.h>

int main()
{
    int num;
    int * nump;

    num = 3;
    nump = &num;

    printf(" num: %d\n", num);
    printf("nump: %p\n", nump);
    printf("*num: %d\n", *nump);

    return 0;
}
```

```
 num: 3
nump: 0x7fff35ca8954
*num: 3
```


5

Pointer and Nonpointer Variables

```c
#include <stdio.h>

int main()
{
    int a[] = {0, 1, 2, 3, 4, 5};
    int * ap;

    printf("a[0]: %d\n", a[0]);
    printf("a[0]: %d\n", *(a));
    printf("a[1]: %d\n", a[1]);
    printf("a[1]: %d\n", *(a+1));
    printf("\n");

    ap = a;
    printf("&a[0]: %p -> a[0]: %d\n", ap, *ap);
    ap++;
    printf("&a[1]: %p -> a[1]: %d\n", ap, *ap);
    ap++;
    printf("&a[2]: %p -> a[2]: %d\n", ap, *ap);

    return 0;
}
```

```
a[0]: 0
a[0]: 0
a[1]: 1
a[1]: 1

&a[0]: 0x7ffd1f1028e0 -> a[0]: 0
&a[1]: 0x7ffd1f1028e4 -> a[1]: 1
&a[2]: 0x7ffd1f1028e8 -> a[2]: 2
```


6

Pointer and Nonpointer Variables

```c
#include <stdio.h>

int main()
{
    int a[] = {0, 1, 2, 3, 4, 5};
    int * ap;
    int i;

    for (i=0; i<6; i++)
        printf("&a[%d]: %p -> a[%d]: %d\n", i, a+i, i, *(a+i));

    printf("\n");

    for (ap=&a[4]; ap>=a; ap--)
        printf("ap: %p -> *ap: %d\n", ap, *ap);

    return 0;
}
```

```
&a[0]: 0x7ffccc77d0b0 -> a[0]: 0
&a[1]: 0x7ffccc77d0b4 -> a[1]: 1
&a[2]: 0x7ffccc77d0b8 -> a[2]: 2
&a[3]: 0x7ffccc77d0bc -> a[3]: 3
&a[4]: 0x7ffccc77d0c0 -> a[4]: 4
&a[5]: 0x7ffccc77d0c4 -> a[5]: 5

ap: 0x7ffccc77d0c0 -> *ap: 4
ap: 0x7ffccc77d0bc -> *ap: 3
ap: 0x7ffccc77d0b8 -> *ap: 2
ap: 0x7ffccc77d0b4 -> *ap: 1
ap: 0x7ffccc77d0b0 -> *ap: 0
```


7

Pointers

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

8

Pointers as Output Parameters

```
1.  #include <stdio.h>
2.
3.  void long_division(int dividend, int divisor, int *quotientp,
4.                     int *remainderp);
5.
6.  int
7.  main(void)
8.  {
9.     int quot, rem;
10.
11.    long_division(40, 3, &quot, &rem);
12.    printf("40 divided by 3 yields quotient %d ", quot);
13.    printf("and remainder %d\n", rem);
14.    return (0);
15. }
16.
17. /*
18.  *  Performs long division of two integers, storing quotient
19.  *  in variable pointed to by quotientp and remainder in
20.  *  variable pointed to by remainderp
21.  */
22. void long_division(int dividend, int divisor, int *quotientp,
23.                     int *remainderp)
24. {
25.    *quotientp = dividend / divisor;
26.    *remainderp = dividend % divisor;
27. }
```


9

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

10

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

11

Dynamic Memory Allocation

- Pointer declaration does not allocate memory for values.

```
double * nump;
```

- Use function malloc to allocate memory

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

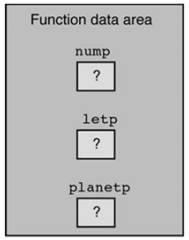
  - Allocates number of bytes defined by the parameter
  - Memory allocated in the heap (not stack)
  - Returns a pointer to the block allocated
  - Memory allocated by malloc could be used to store any value
  - What should be the return type? (type of the pointer)

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


12

Three Pointer-Type Local Variables

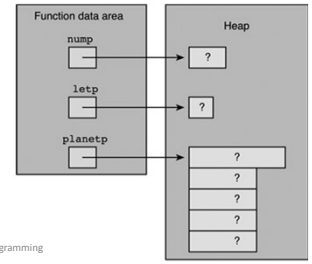
```
int       *nump;
char      *letp;
planet_t  *planetp;
```



13

Dynamic Allocation of Variables

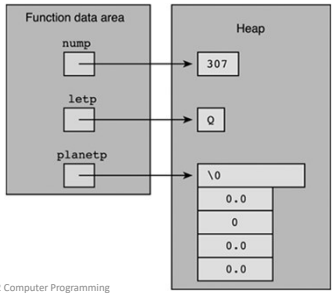
```
nump = (int *)malloc(sizeof(int));
letp = (char *)malloc(sizeof(char));
planetp = (planet_t *)malloc(sizeof(planet_t));
```



14

Assignments to Dynamically Allocated Variables

```
*nump = 307;
*letp = 'Q';
*planetp = blank_planet;
```



15

Components of Dynamically Allocated Structure

- Indirection + component selection

  `(*planetp).name`

- Indirect component selection

  `planetp->name`

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


16

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

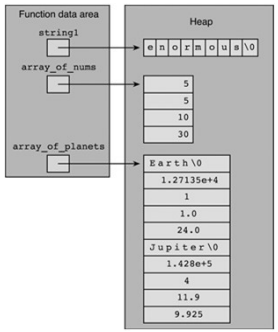
Allocation of Arrays with calloc

```
1.  #include <stdlib.h>  /* gives access to calloc */
2.  int scan_planet(planet_t *plnp);
3.
4.  int
5.  main(void)
6.  {
7.        char     *string1;
8.        int      *array_of_nums;
9.        planet_t *array_of_planets;
10.       int      str_siz, num_nums, num_planets, i;
11.       printf("Enter string length and string> ");
12.       scanf("%d", &str_siz);
13.       string1 = (char *)calloc(str_siz, sizeof (char));
14.       scanf("%s", string1);
15.
16.       printf("\nHow many numbers?> ");
17.       scanf("%d", &num_nums);
18.       array_of_nums = (int *)calloc(num_nums, sizeof (int));
19.       array_of_nums[0] = 5;
20.       for (i = 1;  i < num_nums;  ++i)
21.             array_of_nums[i] = array_of_nums[i - 1] * i;
22.
23.       printf("\nEnter number of planets and planet data> ");
24.       scanf("%d", &num_planets);
25.       array_of_planets = (planet_t *)calloc(num_planets,
26.                                             sizeof (planet_t));
```

(continued)

Allocation of Arrays with calloc

```
27.       for (i = 0;  i < num_planets;  ++i)
28.             scan_planet(&array_of_planets[i]);
29.       . . .
30. }

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
```



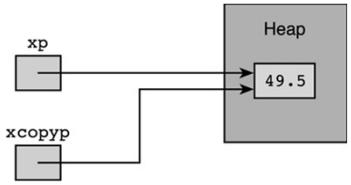
Returning Memory

- 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

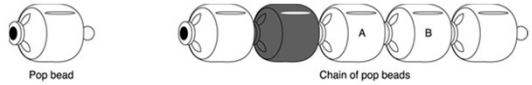
```
double *xp, *xcopyp;
xp = (double *)malloc(sizeof(double));
*xp = 49.5;
xcopyp = xp;
free(xp);
*xcopyp = 52.25;
```



21

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)



22

Dynamic Array Allocation

Local Memory

```
14  void get_many_numbers() {
15      int i, counter = 0, inpt;
16      int * ap, * apb = NULL;
17
18      do {
19          scanf("%d", &inpt);
20          counter++;
21
22          ap = (int *)calloc(counter, sizeof(int));
23          for (i=0;i<counter-1;i++) ap[i] = apb[i];
24          ap[counter-1] = inpt;
25
26          if (apb!=NULL) free(apb);
27          apb = ap;
28      } while (inpt>0);
```

Heap Memory

23

Reminder on typedef and struct

- We use:

```
typedef struct { … } myType;
```

- Tag namespace

```
struct myType { … };
```


24

Reminder on typedef and struct

- We use:

```
typedef struct { … } myType;
```

- Tag namespace

```
struct myType { … };
myType x;
```


25

Reminder on typedef and struct

- We use:

```
typedef struct { … } myType;
```

- Tags – names of structures, unions and enums

```
struct myType { … };
myType x;
```

- Correct version should be

```
struct myType x;
```


26

Reminder on typedef and Struct

- We use:

```
typedef struct { … } myType;
```

- Tag namespace

```
struct myType { … };
myType x;
```

- Need typedef …

```
struct myType { … };
typedef struct myType myType;
```


27

Reminder on typedef and struct

- Or use an abbreviation …

```
typedef struct myType { … } myType;
```

- Or declare an anonymous structure and …

```
typedef struct { … } myType;
```


28

Node Structure

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

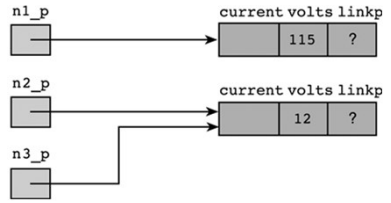
```
typedef struct node_s {
      char            current[3];
      int             volts;
      struct node_s * linkp;
} node_t;
```

  - Use a structure tag

29

Multiple Pointers to the Same Structure

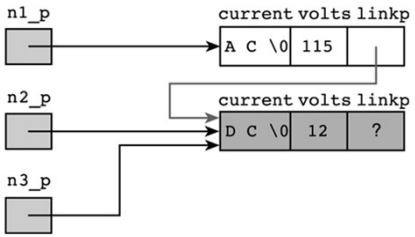
```
node_t *n1_p, *n2_p, *n3_p;
n1_p = (node_t *)malloc(sizeof(node_t));
n1_p->volts = 115;
n2_p = (node_t *)malloc(sizeof(node_t));
n2_p->volts = 12;
n3_p = n2_p;
```



30

Linking Two Nodes

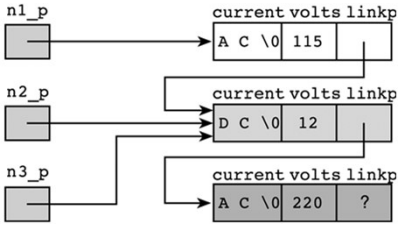
```
n1_p->linkp = n2_p;
```



31

Three-Node Linked List

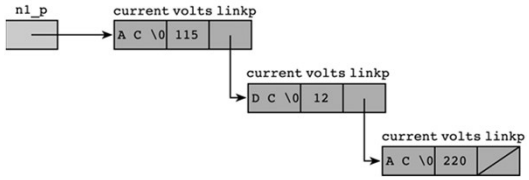
```
n2_p->linkp = (node_t *)malloc(sizeof(node)t));
n2_p->linkp->volts = 220;
strcpy(n2_p->linkp->current, “AC”);
```



32

Three-Element Linked List

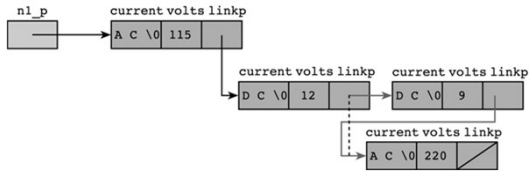
- List head
- End of list indicator
- Empty list



33

Linked List After an Insertion

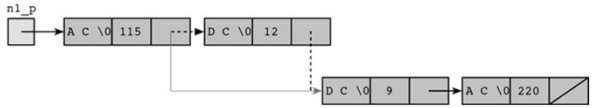
- Linked List can be modified easily
  - Insertion



34

Linked List After a Deletion

- Linked List can be modified easily
  - Insertion
  - Deletion



35

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

36

Recursive function print_list

```
1. /*
2.  *  Displays the list pointed to by headp
3.  */
4. void
5. print_list(list_node_t *headp)
6. {
```


37

Recursive function print_list

```
1.  /*
2.   *  Displays the list pointed to by headp
3.   */
4.  void
5.  print_list(list_node_t *headp)
6.  {
7.      if (headp == NULL) {    /* simple case - an empty list                  */
8.          printf("\n");
9.      } else {                /* recursive step - handles first element       */
10.         printf("%d", headp->digit);  /*      leaves rest to                  */
11.         print_list(headp->restp);    /*      recursion                       */
12.     }
13. }
```


38

Recursive and Iterative List Printing

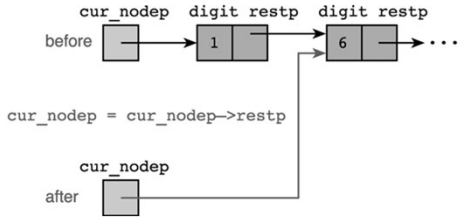
- Tail recursion: recursive call is at the last step
  - Easy to convert it to recursion

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


39

Update of List-Traversing Control Variable



40

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.  {
```


41

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. }
```


42

Recursive 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     */
```


43

Recursive Function get_list

```
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. }
```


44

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.  {
```


45

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. }
```

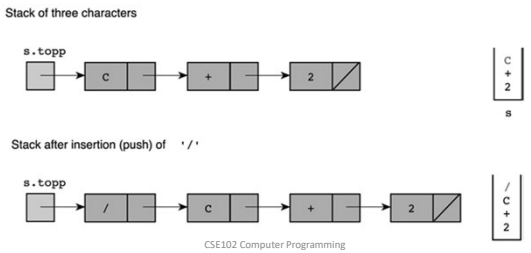

46

Stack

47

Linked List Representation of Stacks

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



48

Structure Types

```
1.  typedef char stack_element_t;
2.
3.  typedef struct stack_node_s {
4.        stack_element_t      element;
5.        struct stack_node_s *restp;
6.  } stack_node_t;
7.
8.  typedef struct {
9.        stack_node_t *topp;
10. } stack_t;
```


Stack Manipulation with push and pop

```
1.  /*
2.   *  Creates and manipulates a stack of characters
3.   */
4.
5.  #include <stdio.h>
6.  #include <stdlib.h>
7.
8.  /*  Include typedefs from Fig. 14.24 */
9.  void push(stack_t *sp, stack_element_t c);
10. stack_element_t pop(stack_t *sp);
```

(continued)

Stack Manipulation with push and pop

```
5.  #include <stdio.h>
6.  #include <stdlib.h>
7.
8.  /*  Include typedefs from Fig. 14.24 */
9.  void push(stack_t *sp, stack_element_t c);
10. stack_element_t pop(stack_t *sp);
```

(continued)

Stack Manipulation with push and pop

```
11. int
12. main(void)
13. {
14.       stack_t s = {NULL};  /* stack of characters - initially empty */
15.
16.       /*  Builds first stack of Fig. 14.23        */
17.       push(&s, '2');
18.       push(&s, '+');
19.       push(&s, 'C');
20.
21.       /*  Completes second stack of Fig. 14.23    */
22.       push(&s, '/');
23.
24.       /*  Empties stack element by element        */
25.       printf("\nEmptying stack: \n");
26.       while (s.topp != NULL) {
27.            printf("%c\n", pop(&s));
28.       }
29.
30.       return (0);
31. }
```

Stack Manipulation with push and pop

```
33. /*
34.  *  The value in c is placed on top of the stack accessed through sp
35.  *  Pre:  the stack is defined
36.  */
37. void
38. push(stack_t        *sp, /* input/output - stack           */
39.      stack_element_t c)  /* input         - element to add */
40. {
41.       stack_node_t *newp;  /* pointer to new stack node  */
42.
43.       /*  Creates and defines new node                   */
44.       newp = (stack_node_t *)malloc(sizeof (stack_node_t));
45.       newp->element = c;
46.       newp->restp = sp->topp;
47.       /*  Sets stack pointer to point to new node       */
48.       sp->topp = newp;
49. }
```


53

Stack Manipulation with push and pop

```
51. /*
52.  *  Removes and frees top node of stack, returning character value
53.  *  stored there.
54.  *  Pre:  the stack is not empty
55.  */
56. stack_element_t
57. pop(stack_t *sp) /* input/output - stack */
58. {
59.       stack_node_t    *to_freep;  /* pointer to node removed */
60.       stack_element_t  ans;       /* value at top of stack   */
61.
62.       to_freep = sp->topp;           /* saves pointer to node being deleted     */
63.       ans = to_freep->element;       /* retrieves value to return               */
64.       sp->topp = to_freep->restp;    /* deletes top node                        */
65.       free(to_freep);                /* deallocates space                       */
66.
67.       return (ans);
68. }
```

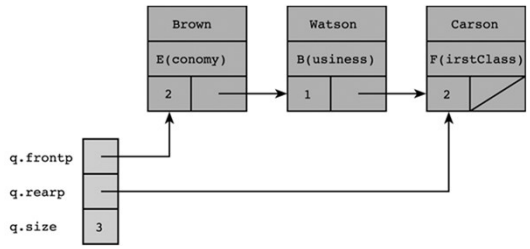

54

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

55

A Queue of Passengers



56

Structure Types

```
1.  /*  Insert typedef for queue_element_t */
2.
3.  typedef struct queue_node_s {
4.        queue_element_t      element;
5.        struct queue_node_s *restp;
6.  } queue_node_t;
7.
8.  typedef struct {
9.       queue_node_t *frontp,
10.                   *rearp;
11.      int           size;
12. } queue_t;
```

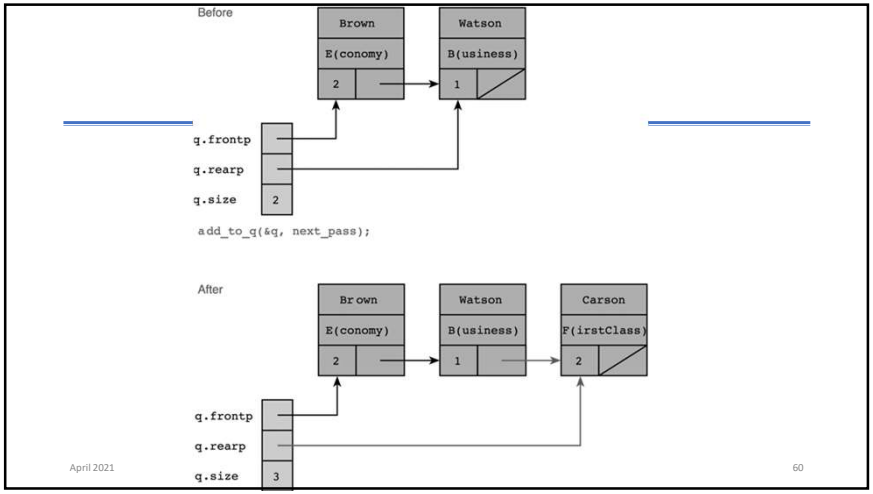

57

```
1.  /*
2.   *  Creates and manipulates a queue of passengers.
3.   */
4.
5.  int scan_passenger(queue_element_t *passp);
6.  void print_passenger(queue_element_t pass);
7.  void add_to_q(queue_t *qp, queue_element_t ele);
8.  queue_element_t remove_from_q(queue_t *qp);
9.  void display_q(queue_t q);
10.
11. int
12. main(void)
13. {
14.       queue_t pass_q = {NULL, NULL, 0}; /* passenger queue - initialized to
15.                                            empty state */
16.       queue_element_t next_pass, fst_pass;
17.       char choice;  /* user's request */
18.
19.       /*  Processes requests  */
20.       do {
21.           printf("Enter A(dd), R(emove), D(isplay), or Q(uit)> ");
22.           scanf(" %c", &choice);
23.           switch (toupper(choice)) {
24.           case 'A':
25.                 printf("Enter passenger data> ");
26.                 scan_passenger(&next_pass);
27.                 add_to_q(&pass_q, next_pass);
28.                 break;
29.
```


58

```
30.           case 'R':
31.                 if (pass_q.size > 0) {
32.                       fst_pass = remove_from_q(&pass_q);
33.                       printf("Passenger removed from queue: \n");
34.                       print_passenger(fst_pass);
35.                 } else {
36.                       printf("Queue empty - noone to delete\n");
37.                 }
38.                 break;
39.
40.           case 'D':
41.                 if (pass_q.size > 0)
42.                       display_q(pass_q);
43.                 else
44.                       printf("Queue is empty\n");
45.                 break;
46.
47.           case 'Q':
48.                 printf("Leaving passenger queue program with %d \n",
49.                        pass_q.size);
50.                 printf("passengers in the queue\n");
51.                 break;
52.
53.           default:
54.                 printf("Invalid choice -- try again\n");
55.           }
56.       } while (toupper(choice) != 'Q');
57.
58.       return (0);
59. }
```

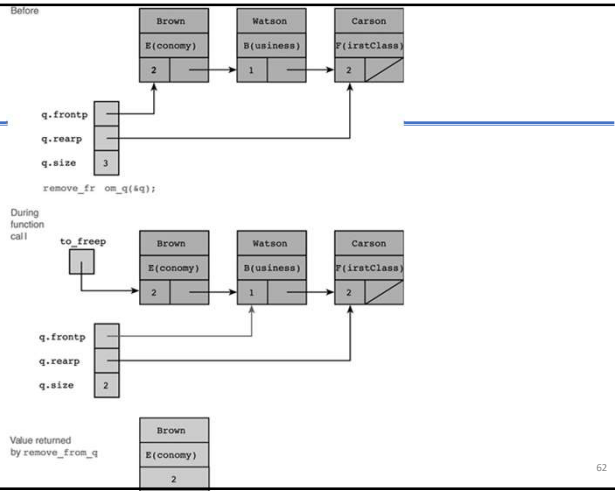

59



60

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


61



62

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


63

Case Study: Ordered Lists

The position of elements is determined by their key value

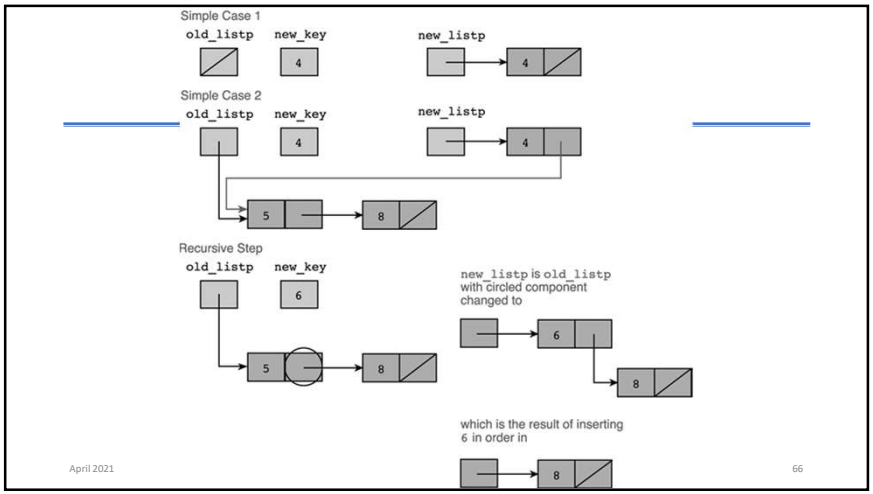
- Increasing or decreasing order

```
/*
 *  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
```


64

```c
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);
}
```


65



66

```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);
}
```


67

```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);
}
```


68

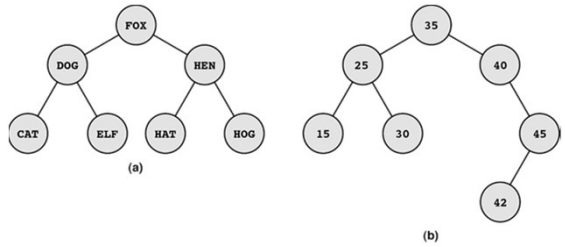
```
1.  /*
2.   *  Deletes first node containing the target key from an ordered list.
3.   *  Returns 1 if target found and deleted, 0 otherwise.
4.   */
5.  int
6.  delete(ordered_list_t *listp,  /* input/output - ordered list   */
7.         int            target)  /* input - key of node to delete */
8.  {
9.        int is_deleted;
10.
11.       listp->headp = delete_ordered_node(listp->headp, target,
12.                                          &is_deleted);
13.       if (is_deleted)
14.             --(listp->size);
15.
16.       return (is_deleted);
17. }
```


69

```
1.  /*
2.   *  If possible, deletes node containing target key from list whose first
3.   *  node is pointed to by listp, returning pointer to modified list and
4.   *  freeing deleted node.  Sets output parameter flag to indicate whether or
5.   *  not deletion occurred.
6.   */
7.  list_node_t *
8.  delete_ordered_node(list_node_t *listp,      /* input/output - list to modify */
9.                      int         target,     /* input - key of node to delete */
10.                     int        *is_deletedp) /* output - flag indicating
11.                                                  whether or not target node
12.                                                  found and deleted          */
13. {
14.       list_node_t *to_freep, *ansp;
15.
16.       /* if list is empty - can't find target node       - simple case 1      */
17.       if (listp == NULL) {
18.             *is_deletedp = 0;
19.             ansp = NULL;
20.
21.       /* if first node is the target, delete it          - simple case 2      */
22.       } else if (listp->key == target) {
23.             *is_deletedp = 1;
24.             to_freep = listp;
25.             ansp = listp->restp;
26.             free(to_freep);
27.
28.       /* if past the target value, give up               - simple case 3      */
29.       } else if (listp->key > target) {
30.             *is_deletedp = 0;
31.             ansp = listp;
32.
33.       /* in case target node is farther down the list,   - recursive step
34.          have recursive call modify rest of list and then return list        */
35.       } else {
36.             ansp = listp;
37.             ansp->restp = delete_ordered_node(listp->restp, target,
38.                                               is_deletedp);
39.       }
40.
41.       return (ansp);
42. }
```


70

Binary Trees



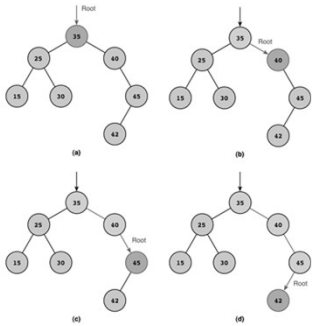
71

Binary Search Trees

- How to implement?

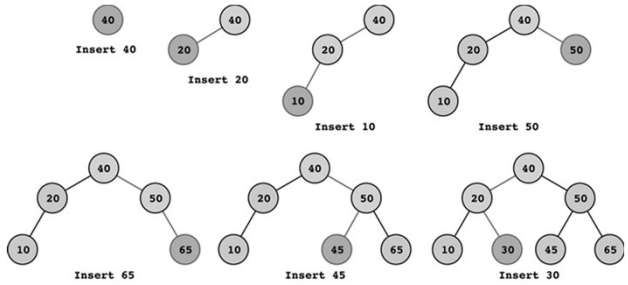
72

Binary Tree Search for 42



73

Building a Binary Search Tree



74

Creating a Binary Search Tree

```
1.  /*
2.   *  Create and display a binary search tree of integer keys.
3.   */
4.
5.  #include <stdio.h>
6.  #include <stdlib.h>
7.
8.  #define TYPED_ALLOC(type) (type *)malloc(sizeof (type))
9.
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.
```


75

```
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)

76

```
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. }
```


77

Thanks for listening!

78