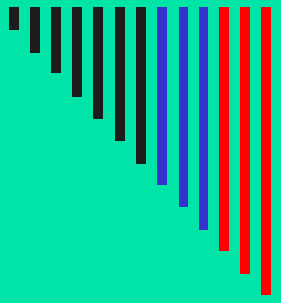


Chapter 7

Binary Search Trees

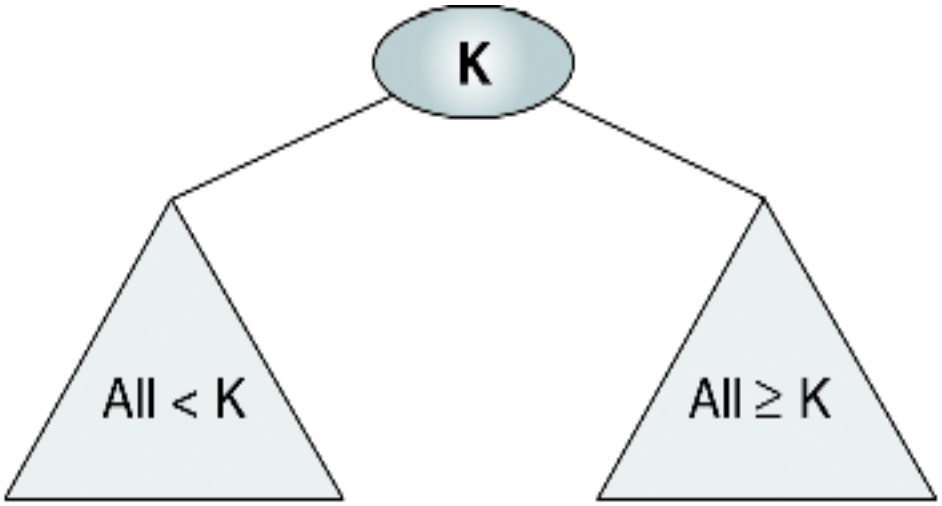
Objectives

Upon completion you will be able to:

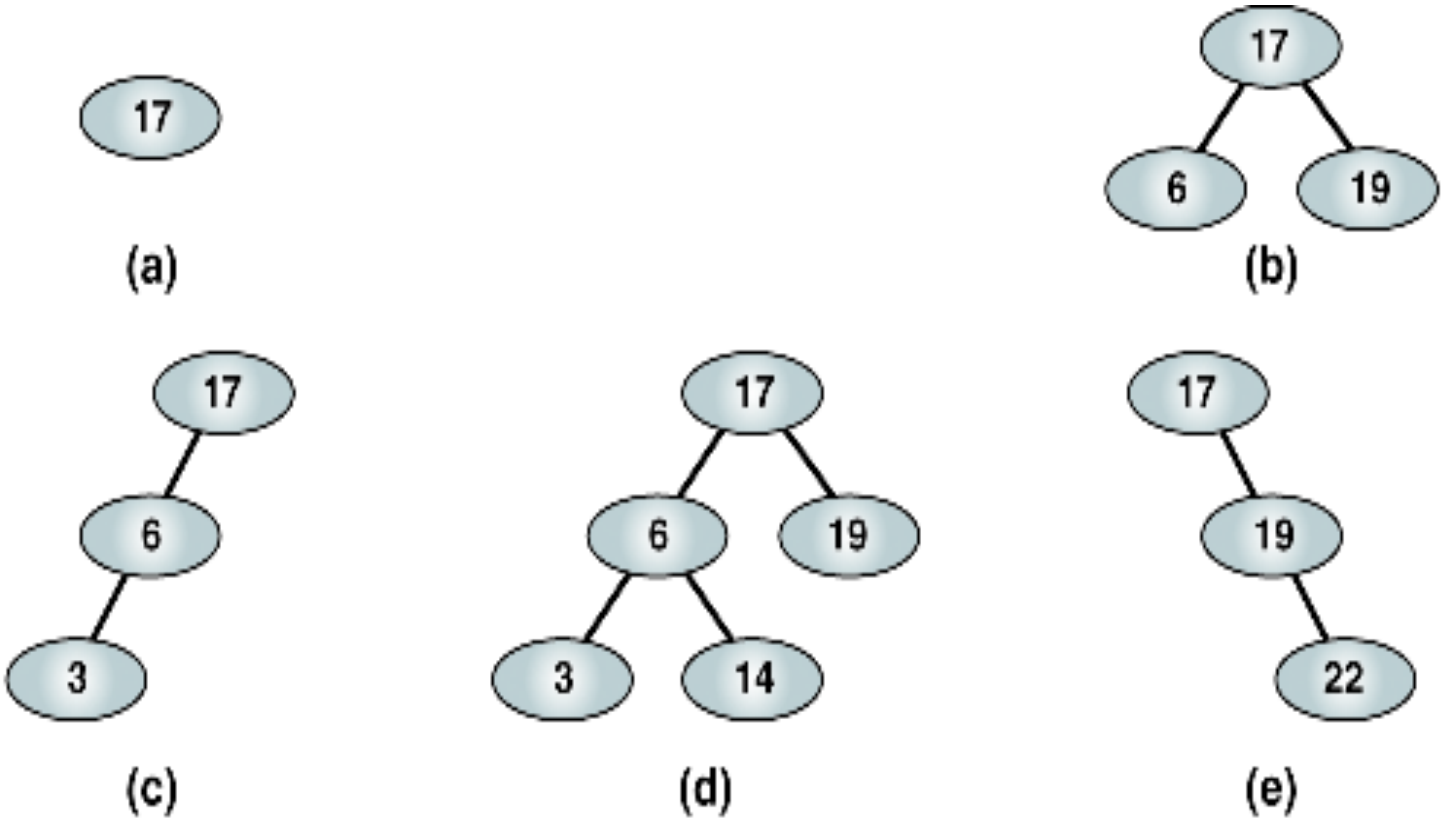
- Create and implement binary search trees
- Understand the operation of the binary search tree ADT
- Write application programs using the binary search tree ADT
- Design and implement a list using a BST
- Design and implement threaded trees

7-1 Basic Concepts

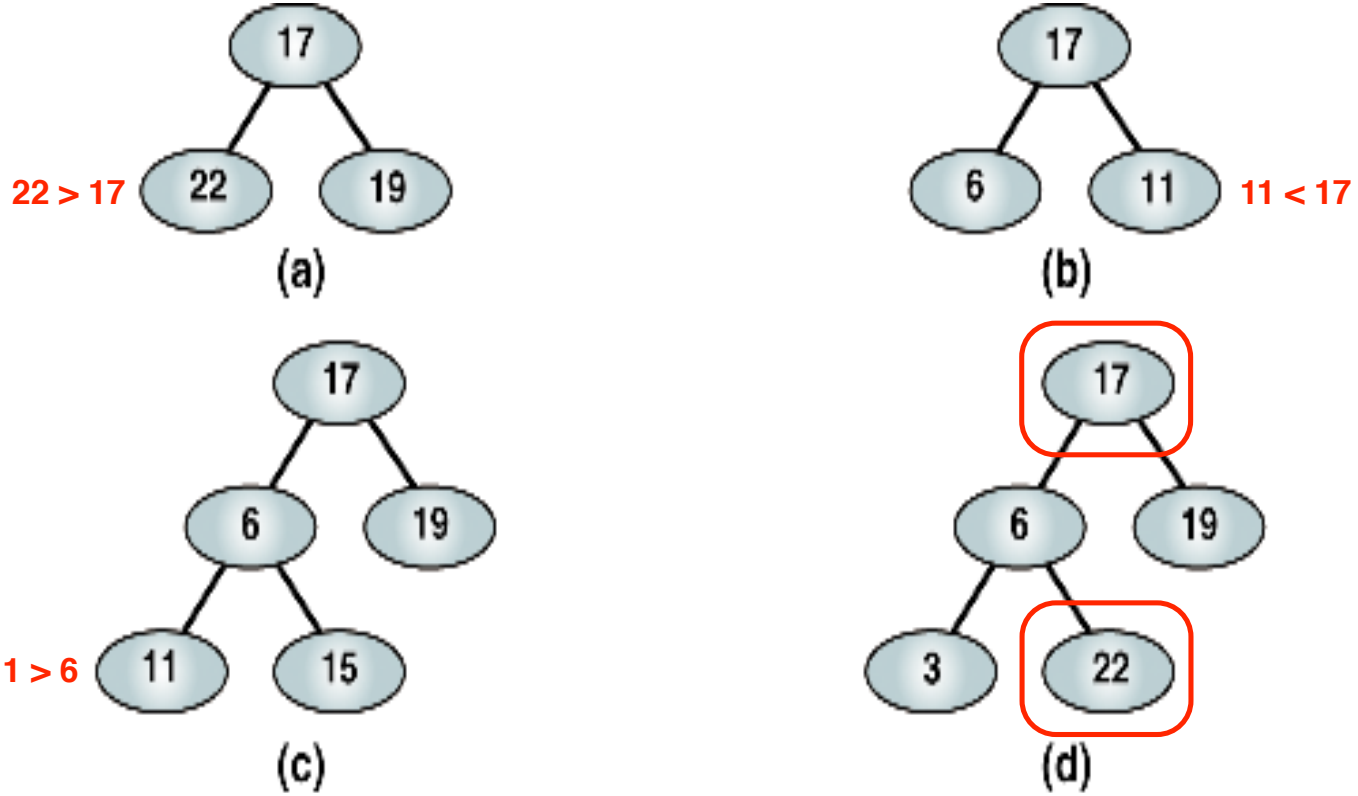
Binary search trees provide an excellent structure for searching a list and at the same time for inserting and deleting data into the list.



Valid Binary Search Tree



Invalid Binary Search Tree

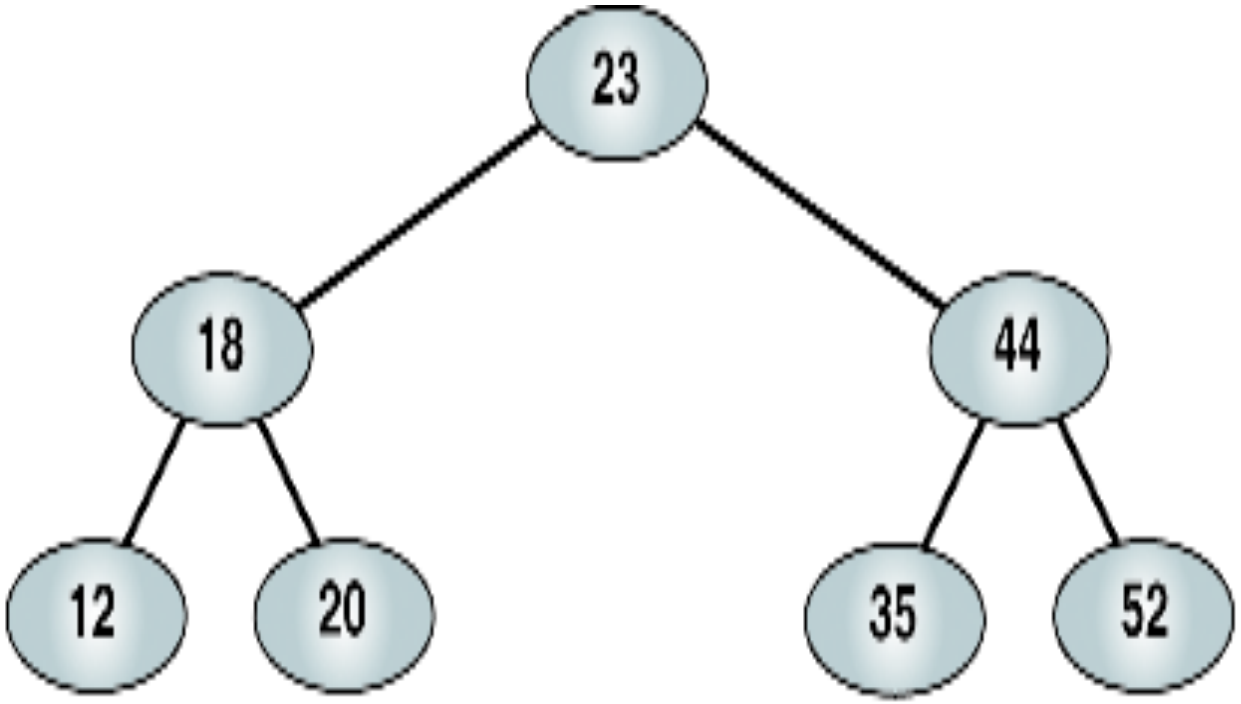


7-2 BST Operations

We discuss four basic BST operations: traversal, search, insert, and delete; and develop algorithms for searches, insertion, and deletion.

- **Traversals**
- **Searches**
- **Insertion**
- **Deletion**

Example of Binary Search Tree



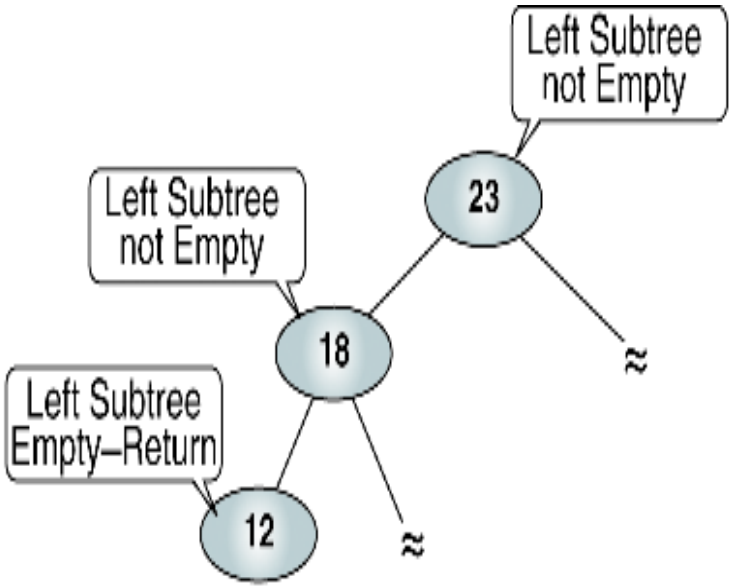
Preorder: 23 18 12 20 44 35 52 root -> left -> right

Postorder: 12 20 18 35 52 44 23

Inorder: 12 18 20 23 35 44 52

對Binary Search Tree做Inorder Traversals相當於從小到大(increasing)做sorting

Find Smallest Node in a BST




```
Algorithm findSmallestBST (root)
This algorithm finds the smallest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of smallest node
1  if (left subtree empty)
   1   return (root)
2  end if
3  return findSmallestBST (left subtree)
end findSmallestBST
```

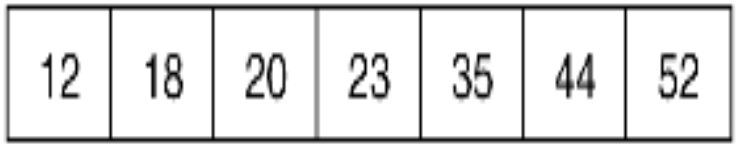
Find Largest Node in a BST

```
Algorithm findLargestBST (root)
This algorithm finds the largest node in a BST.
   Pre    root is a pointer to a nonempty BST or subtree
   Return address of largest node returned
1  if (right subtree empty)
   1   return (root)
2  end if
3  return findLargestBST (right subtree)
end findLargestBST
```

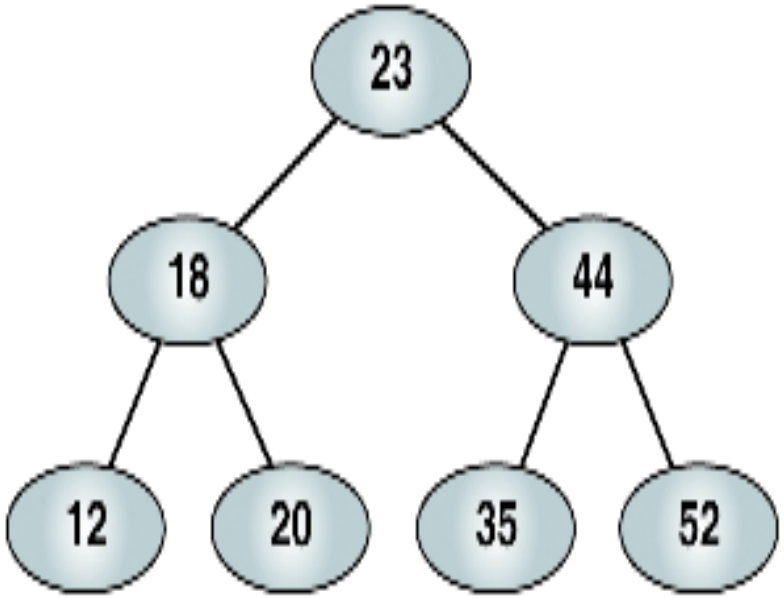
BST and Binary Search

Time complexity of searching

Sequenced array



O(n)



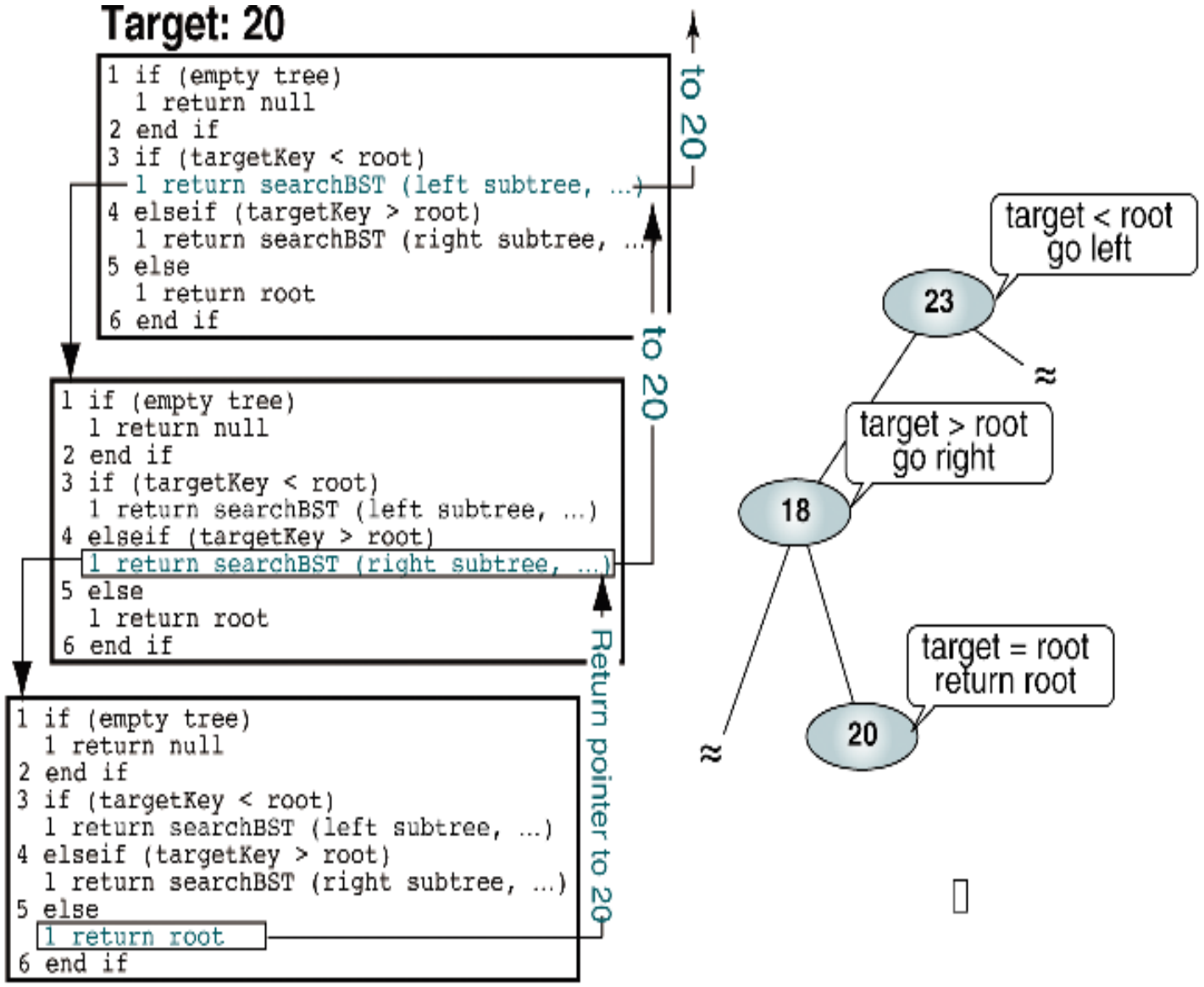
O(logn)

Search points in binary search

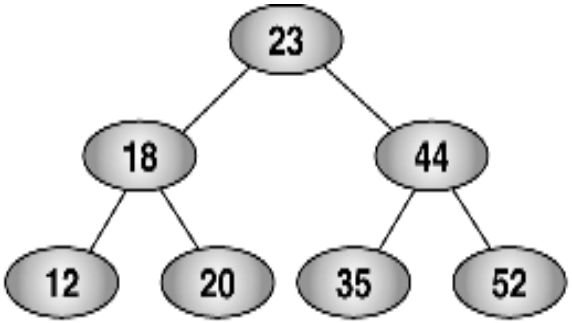
Search BST

```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
   Pre    root is the root to a binary tree or subtree
          targetKey is the key value requested
   Return the node address if the value is found
          null if the node is not in the tree
1  if (empty tree)
      Not found
   1  return null
2  end if
3  if (targetKey < root)
   1  return searchBST (left subtree, targetKey)
4  else if (targetKey > root)
   1  return searchBST (right subtree, targetKey)
5  else
      Found target key
   1  return root
6  end if
end searchBST
```

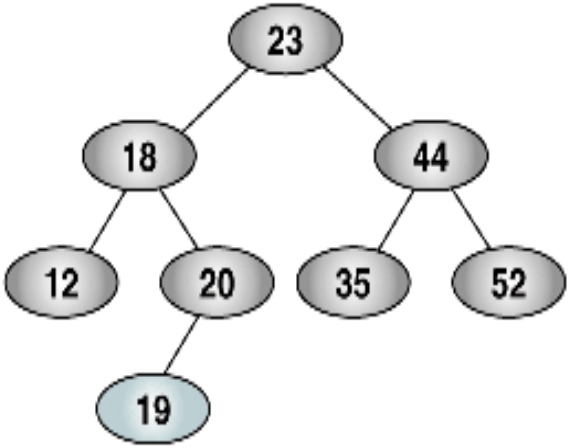
Searching a BST



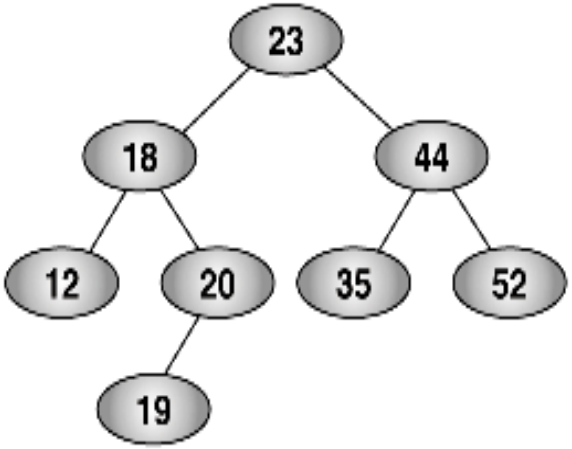
BST Insertion



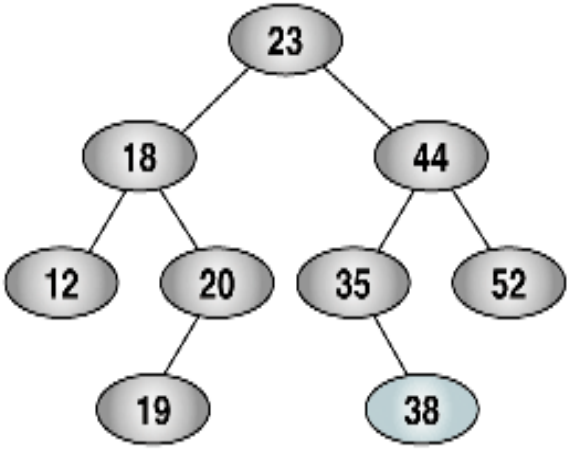
(a) Before inserting 19



(b) After inserting 19



(c) Before inserting 38

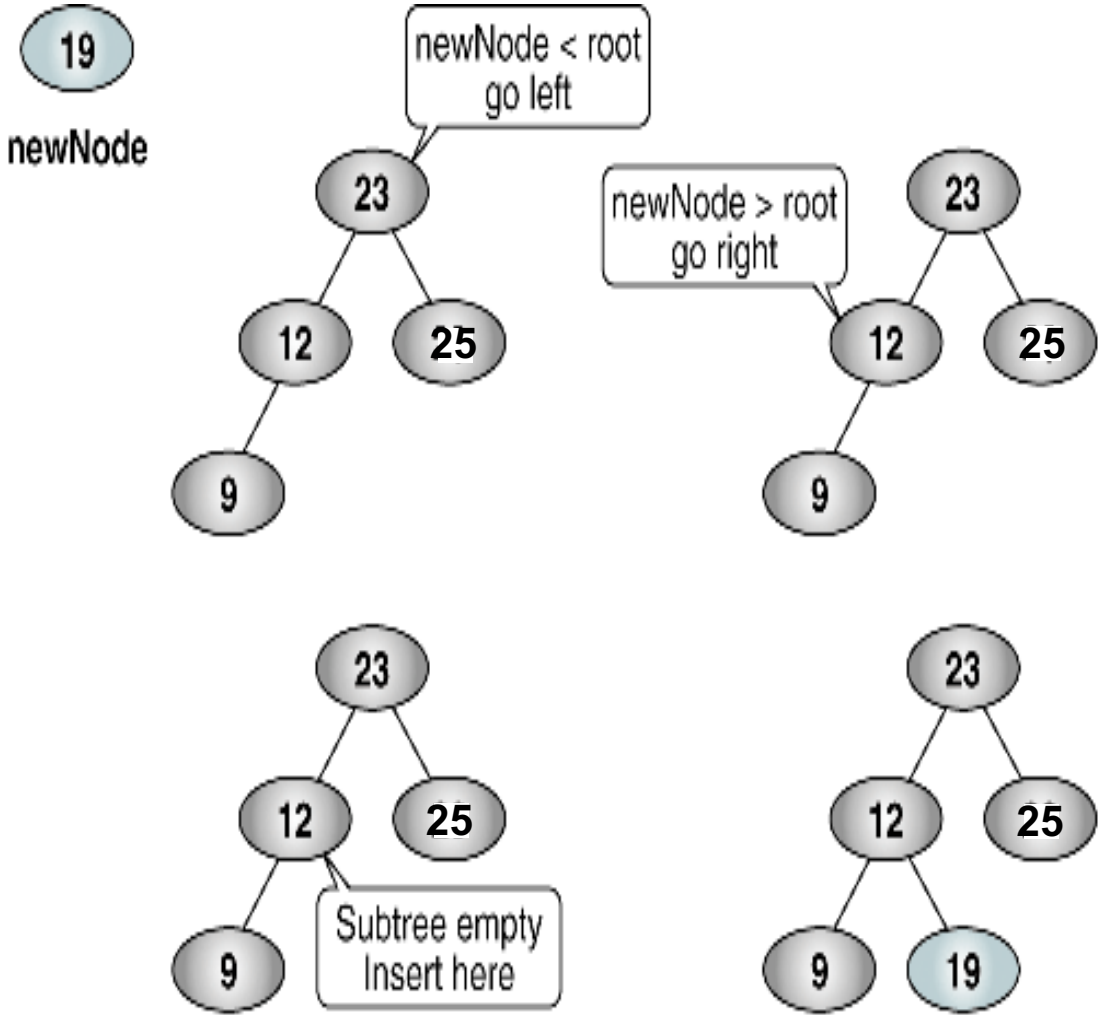


(d) After inserting 38

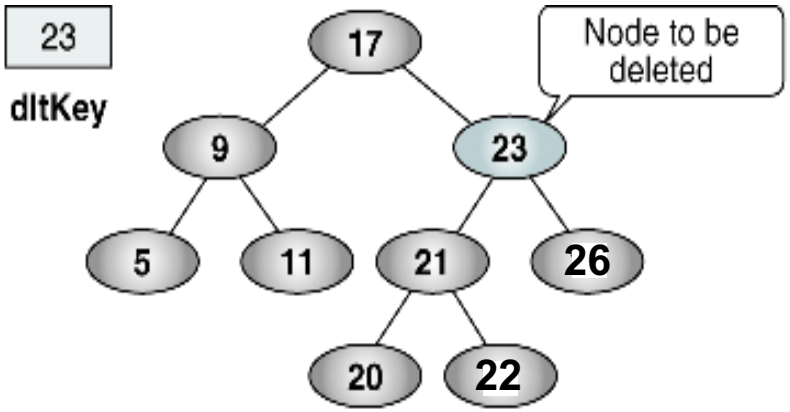
Add Node to BST

```
Algorithm addBST (root, newNode)
Insert node containing new data into BST using recursion.
   Pre    root is address of current node in a BST
          newNode is address of node containing data
   Post   newNode inserted into the tree
   Return address of potential new tree root
1  if (empty tree)
   1   set root to newNode
   2   return newNode
2  end if
   Locate null subtree for insertion
3  if (newNode < root)
   1   return addBST (left subtree, newNode)
4  else
   1   return addBST (right subtree, newNode)
5  end if
end addBST
```

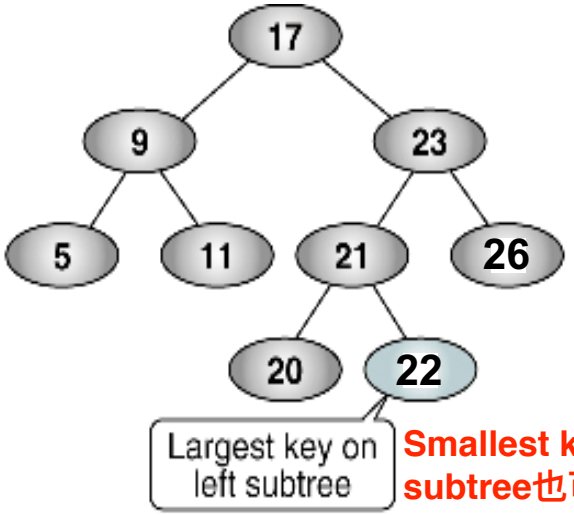
Trace of Recursive BST Insert



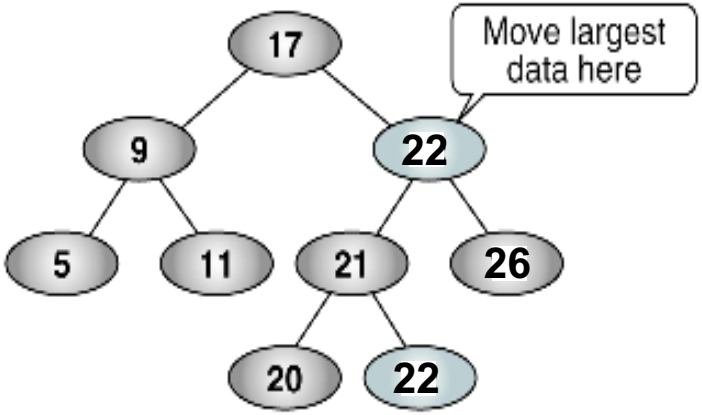
Delete BST Test Cases



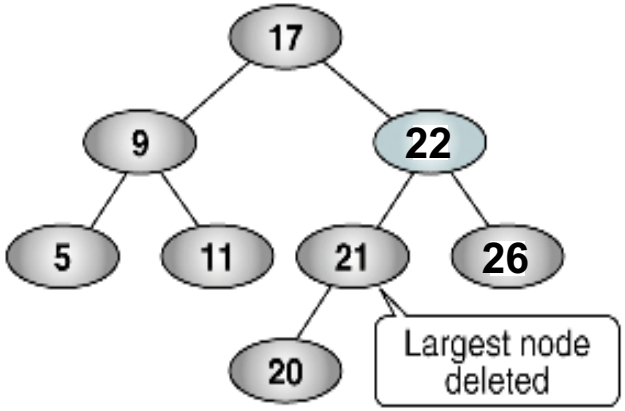
(a) Find dltKey



(b) Find largest



(c) Move largest data



(d) Delete largest node

Delete Node From BST

```
Algorithm deleteBST (root, dltKey)
This algorithm deletes a node from a BST.
   Pre    root is reference to node to be deleted
          dltKey is key of node to be deleted
   Post   node deleted
          if dltKey not found, root unchanged
   Return true if node deleted, false if not found
1  if (empty tree)
   1   return false
2  end if
3  if (dltKey < root)
   1   return deleteBST (left subtree, dltKey)
4  else if (dltKey > root)
   1   return deleteBST (right subtree, dltKey)
5  else
       Delete node found--test for leaf node
   1   If (no left subtree)
       1   make right subtree the root
       2   return true
```

Delete Node From BST (cont.)

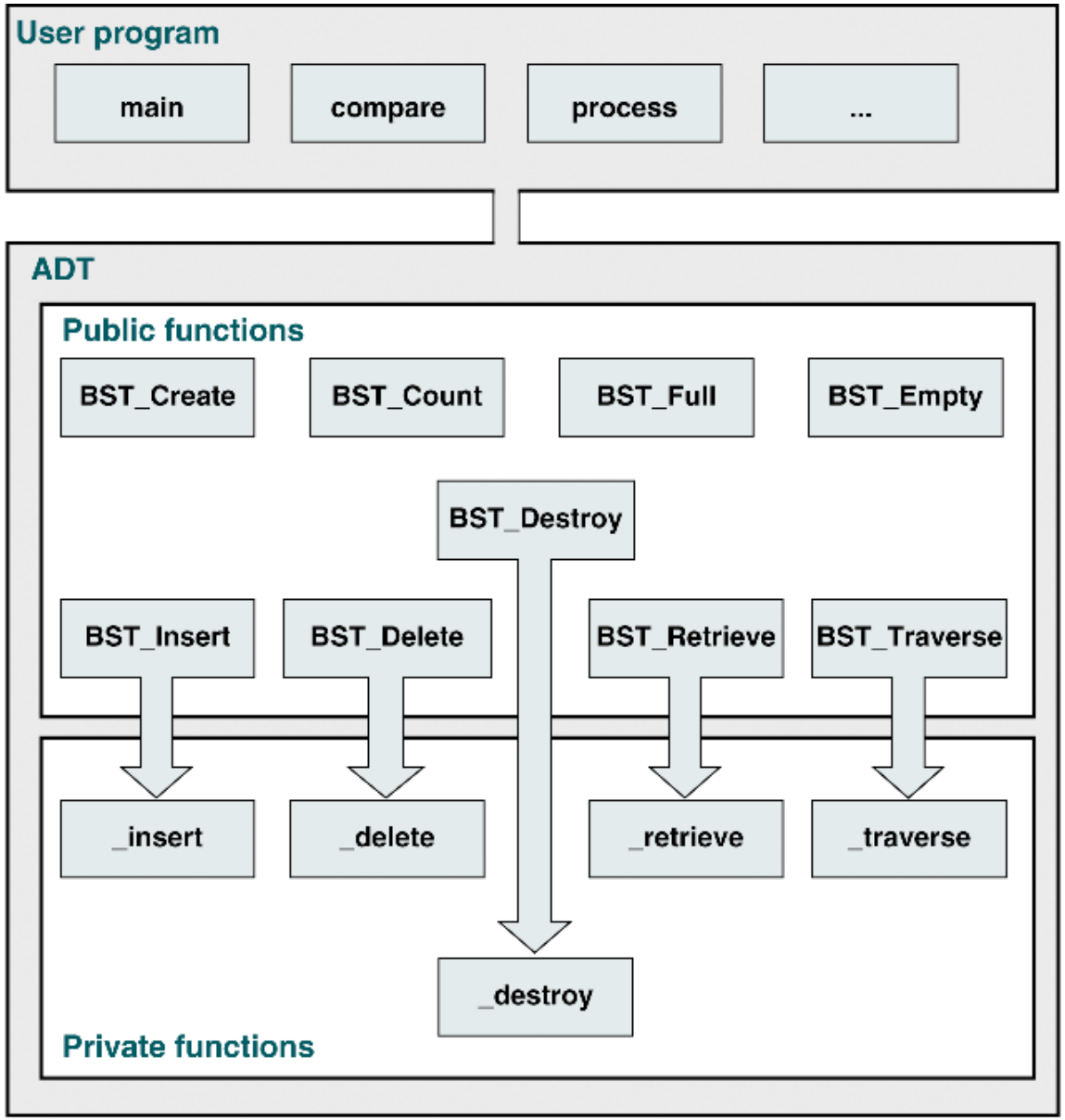
```
   2  else if (no right subtree)
      1  make left subtree the root
      2  return true
   3  else
         Node to be deleted not a leaf. Find largest node on
         left subtree.
      1  save root in deleteNode
      2  set largest to largestBST (left subtree)
      3  move data in largest to deleteNode
      4  return deleteBST (left subtree of deleteNode,
                           key of largest
   4  end if
 6 end if
end deleteBST
```

7-3 Binary Search Tree ADT

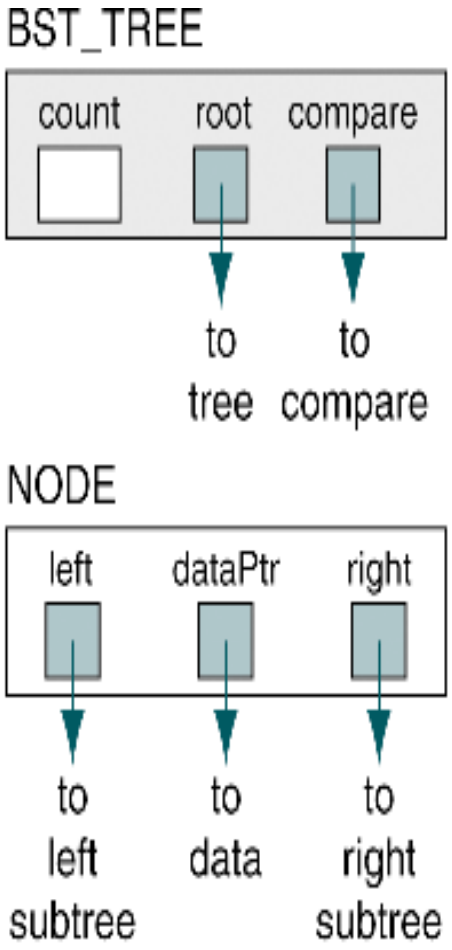
We begin this section with a discussion of the BST data structure and write the header file for the ADT. We then develop 14 programs that we include in the ADT.

- Data Structure
- Algorithms

BST ADT Design



BST Tree Data Structure




```
typedef struct
  {
   int count;
   int  (*compare)
         (void* argu1,
          void* argu2);
   NODE* root;
  } BST_TREE;

typedef struct node
  {
   void*        dataPtr;
   struct node* left;
   struct node* right;
  } NODE;
```

BST Declarations

```
1  /* Header file for binary search tree (BST). Contains
2     structural definitions and prototypes for BST.
3        Written by:
4        Date:
5  */
6  #include <stdbool.h>
7
8  // Structure Declarations
9  typedef struct node
10    {
11     void*        dataPtr;
12     struct node* left;
13     struct node* right;
14    } NODE;
15
```

BST Declarations (cont.)

```
16  typedef struct
17     {
18      int   count;
19      int  (*compare) (void* argu1, void* argu2);
20      NODE*  root;
21     } BST_TREE;
22
23  // Prototype Declarations
24     BST_TREE* BST_Create
25              (int (*compare) (void* argu1, void* argu2));
26     BST_TREE* BST_Destroy (BST_TREE* tree);
27
28     bool  BST_Insert   (BST_TREE* tree, void* dataPtr);
29     bool  BST_Delete   (BST_TREE* tree, void* dltKey);
30     void* BST_Retrieve (BST_TREE* tree, void* keyPtr);
31     void  BST_Traverse (BST_TREE* tree,
32                         void (*process)(void* dataPtr));
```

BST Declarations (cont.)

```
33
34      bool BST_Empty (BST_TREE* tree);
35      bool BST_Full  (BST_TREE* tree);
36      int  BST_Count (BST_TREE* tree);
37
38      static NODE* _insert
39                     (BST_TREE* tree, NODE* root,
40                      NODE* newPtr);
41      static NODE* _delete
42                     (BST_TREE* tree,    NODE* root,
43                      void*     dataPtr, bool* success);
44      static void* _retrieve
45                     (BST_TREE* tree,
46                      void* dataPtr, NODE* root);
47      static void _traverse
48                     (NODE* root,
49                      void (*process) (void* dataPtr));
50      static void _destroy (NODE* root);
```

Create BST Application interface

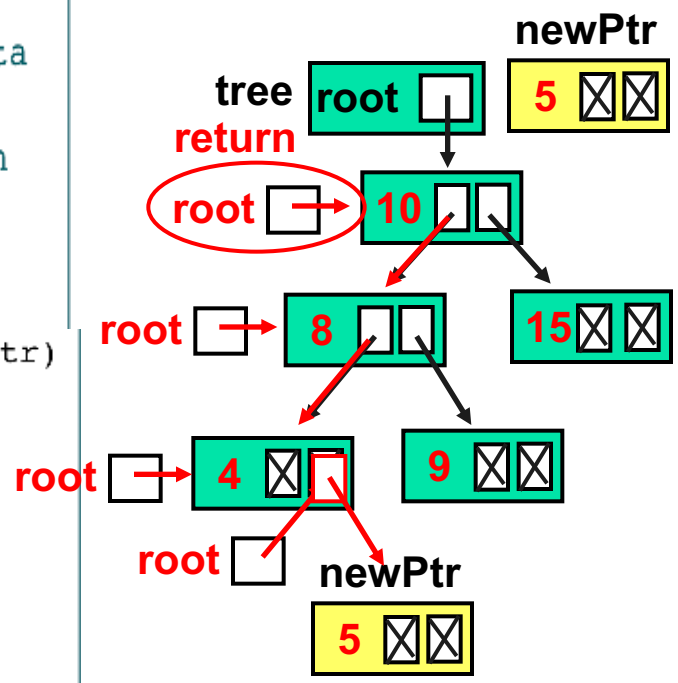
```
1  /* ================ BST_Create ================
2     Allocates dynamic memory for an BST tree head
3     node and returns its address to caller
4        Pre    compare is address of compare function
5               used when two nodes need to be compared
6        Post   head allocated or error returned
7        Return head node pointer; null if overflow
8  */
9  BST_TREE* BST_Create
10          (int  (*compare) (void* argu1, void* argu2))
11 {
12 // Local Definitions
13    BST_TREE* tree;
14
15 // Statements
16    tree = (BST_TREE*) malloc (sizeof (BST_TREE));
17    if (tree)
18       {
19        tree->root    = NULL;
20        tree->count   = 0;
21        tree->compare = compare;
22       } // if
23
24    return tree;
25 }  // BST_Create
```

Insert BST Application Interface

```
 1  /* ================= BST_Insert ===================
 2     This function inserts new data into the tree.
 3        Pre    tree is pointer to BST tree structure
 4        Post   data inserted or memory overflow
 5        Return Success (true) or Overflow (false)
 7  bool BST_Insert (BST_TREE* tree, void* dataPtr)
 8  {
 9  // Local Definitions
10     NODE* newPtr;
11
12  // Statements
13     newPtr = (NODE*)malloc(sizeof(NODE));
14     if (!newPtr)
15        return false;
16
17     newPtr->right   = NULL;
18     newPtr->left    = NULL;
19     newPtr->dataPtr = dataPtr;
20
21     if (tree->count == 0)
22         tree->root  =  newPtr;
23     else
24         _insert(tree, tree->root, newPtr);
25
26     (tree->count)++;
27     return true;
28  }  // BST_Insert
```

Internal Insert Function

```
1  /* ==================== _insert ====================
2     This function uses recursion to insert the new data
3     into a leaf node in the BST tree.
4        Pre    Application has called BST_Insert, which
5               passes root and data pointer
6        Post   Data have been inserted
7        Return pointer to [potentially] new root
9  NODE* _insert (BST_TREE* tree, NODE* root, NODE* newPtr)
10 {
11 // Statements
12    if (!root)
13       // if NULL tree
14       return newPtr;
15
16    // Locate null subtree for insertion
17    if (tree->compare(newPtr->dataPtr,
18                      root->dataPtr) < 0)
19       {
20        root->left = _insert(tree, root->left, newPtr);
21        return root;
22       } // new < node
23    else
24       // new data >= root data
25       {
26        root->right = _insert(tree, root->right, newPtr);
27        return root;
28         } // else new data >= root data
29    return root;
30 } // _insert
```



Delete BST Application Interface

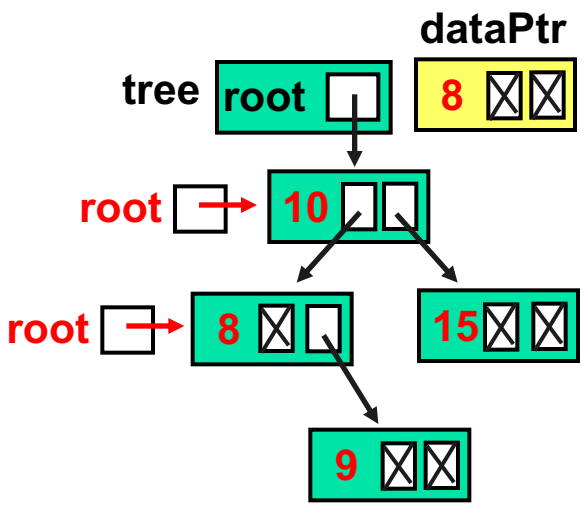
```
11  bool BST_Delete (BST_TREE* tree, void* dltKey)
12  {
13  // Local Definitions
14     bool  success;
15     NODE* newRoot;
16
17  // Statements
18     newRoot = _delete (tree, tree->root, dltKey,
19                        &success);
20     if (success)
21        {
22         tree->root = newRoot;
23         (tree->count)--;
24         if (tree->count == 0)
25             // Tree now empty
26             tree->root = NULL;
27        } // if
28     return success;
29  }  // BST_Delete
```

Internal Delete Function

```
1  /* ==================== _delete ====================
2     Deletes node from the tree and rebalances
3     tree if necessary.
4        Pre    tree initialized--null tree is OK
5               dataPtr contains key of node to be deleted
6        Post   node is deleted and its space recycled
7               -or- if key not found, tree is unchanged
8              success is true if deleted; false if not
9       Return pointer to root
```

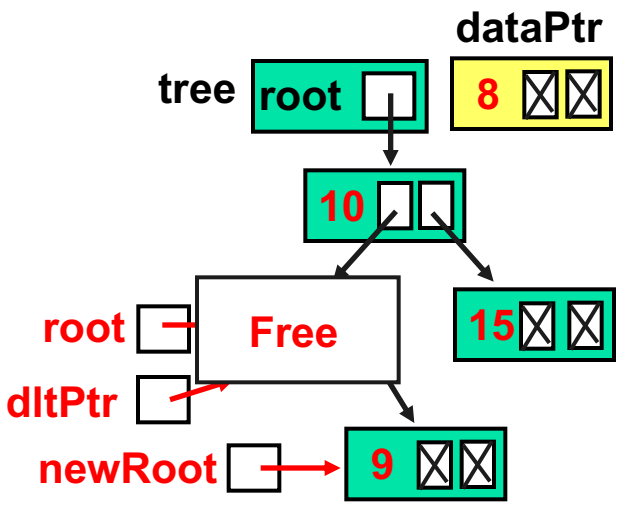
Internal Delete Function (cont.)

```
10  */
11  NODE*  _delete (BST_TREE* tree,    NODE* root,
12                  void*     dataPtr, bool* success)
13  {
14  // Local Definitions
15     NODE* dltPtr;
16     NODE* exchPtr;
17     NODE* newRoot;
18     void* holdPtr;
19
20  // Statements
21     if (!root)
22        {
23         *success = false;
24         return NULL;
25        } // if
26
27     if (tree->compare(dataPtr, root->dataPtr) < 0)
28           root->left  = _delete (tree,    root->left,
29                                  dataPtr, success);
30     else if (tree->compare(dataPtr, root->dataPtr) > 0)
31           root->right = _delete (tree,    root->right,
32                                  dataPtr, success);
```



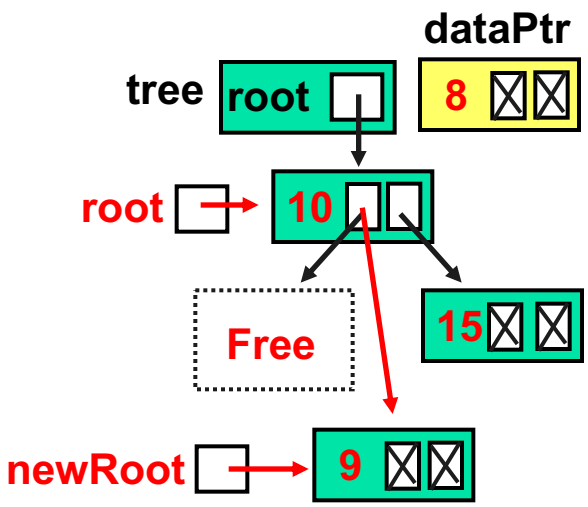
Internal Delete Function (cont.)

```
33      else
34           // Delete node found--test for leaf node
35           {
36            dltPtr = root;
37            if (!root->left)
38                 // No left subtree
39                 {
40                  free (root->dataPtr);
41                  newRoot = root->right;
42                  free (dltPtr);
43                  *success = true;
44                  return newRoot;
45                 } // if true
46            else
47                 if (!root->right)
48                      // Only left subtree
49                      {
50                       newRoot = root->left;
51                       free (dltPtr);
52                       *success = true;
53                       return newRoot;          // base case
54                      } // if
55                 else
56                      // Delete Node has two subtrees
57                      {
```



Internal Delete Function (cont.)

```
10  */
11  NODE*   _delete (BST_TREE* tree,     NODE* root,
12                   void*     dataPtr, bool* success)
13  {
14  // Local Definitions
15     NODE* dltPtr;
16     NODE* exchPtr;
17     NODE* newRoot;
18     void* holdPtr;
19
20  // Statements
21     if (!root)
22        {
23         *success = false;
24         return NULL;
25        } // if
26
27     if (tree->compare(dataPtr, root->dataPtr) < 0)
28           root->left  = _delete (tree,     root->left,
29                                  dataPtr, success);
30     else if (tree->compare(dataPtr, root->dataPtr) > 0)
31           root->right = _delete (tree,     root->right,
32                                  dataPtr, success);
```



Internal Delete Function (cont.)

```
58                      exchPtr = root->left;
59                      // Find largest node on left subtree
60                      while (exchPtr->right)
61                          exchPtr = exchPtr->right;
62
63                      // Exchange Data
64                      holdPtr          = root->dataPtr;
65                      root->dataPtr    = exchPtr->dataPtr;
66                      exchPtr->dataPtr = holdPtr;
67                      root->left       =
68                          _delete (tree,   root->left,
69                                   exchPtr->dataPtr, success);
70                  } // else
71          } // node found
72      return root;
73  } // _delete
```

Redundant !

Retrieve BST Application Interface

```
1  /* ==================== BST_Retrieve ====================
2     Retrieve node searches tree for the node containing
3     the requested key and returns pointer to its data.
4        Pre     Tree has been created (may be null)
5                data is pointer to data structure
6                     containing key to be located
7        Post    Tree searched and data pointer returned
8        Return  Address of matching node returned
9                If not found, NULL returned
10 */
11 void* BST_Retrieve  (BST_TREE* tree, void* keyPtr)
12 {
13 // Statements
14    if (tree->root)
15        return _retrieve (tree, keyPtr, tree->root);
16    else
17        return NULL;
18 }  // BST_Retrieve
```

Internal Retrieve Function

```
1  /* ==================== _retrieve ====================
2     Searches tree for node containing requested key
3     and returns its data to the calling function.
4        Pre     _retrieve passes tree, dataPtr, root
5                dataPtr is pointer to data structure
6                   containing key to be located
7        Post    tree searched; data pointer returned
8        Return  Address of data in matching node
9                If not found, NULL returned
10 */
11 void* _retrieve (BST_TREE* tree,
12                   void* dataPtr, NODE* root)
13 {
14 // Statements
15    if (root)
```

Internal Retrieve Function

```
16          {
17           if (tree->compare(dataPtr, root->dataPtr) < 0)
18              return _retrieve(tree, dataPtr, root->left);
19          else if (tree->compare(dataPtr,
20                   root->dataPtr) > 0)
21             return _retrieve(tree, dataPtr, root->right);
22           else
23              // Found equal key
24              return root->dataPtr;
25          } // if root
26       else
27          // Data not in tree
28          return NULL;
29    } // _retrieve
```

Traverse BST Application Interface

```
1  /* ================== BST_Traverse ==================
2     Process tree using inorder traversal.
3        Pre   Tree has been created (may be null)
4              process "visits" nodes during traversal
5        Post  Nodes processed in LNR (inorder) sequence
6  */
7  void BST_Traverse (BST_TREE* tree,
8                     void (*process) (void* dataPtr))
9  {
10 // Statements
11    _traverse (tree->root, process);
12    return;
13 }  // end BST_Traverse
```

Internal Traverse Function

```
1  /* ==================== _traverse ====================
2     Inorder tree traversal. To process a node, we use
3     the function passed when traversal was called.
4        Pre   Tree has been created (may be null)
5        Post  All nodes processed
6  */
7  void _traverse (NODE* root,
8                  void (*process) (void* dataPtr))
9  {
10 // Statements
11 if  (root)
12     {
13      _traverse (root->left, process);
14      process   (root->dataPtr);
15      _traverse (root->right, process);
16     } // if
17 return;
18 }  // _traverse
```

Empty BST Application Interface

```
1  /* ================== BST_Empty =================
2     Returns true if tree is empty; false if any data.
3        Pre      Tree has been created. (May be null)
4        Returns  True if tree empty, false if any data
5  */
6  bool BST_Empty (BST_TREE* tree)
7  {
8  // Statements
9     return (tree->count == 0);
10 }  // BST_Empty
```

Full BST Application Interface

```
1  /* ===================== BST_Full ====================
2     If there is no room for another node, returns true.
3        Pre      tree has been created
4        Returns  true if no room for another insert
5                 false if room
6  */
7  bool BST_Full (BST_TREE* tree)
8  {
9  // Local Definitions
10    NODE* newPtr;
11
12 // Statements
13    newPtr = (NODE*)malloc(sizeof (*(tree->root)));
14    if (newPtr)
15       {
16        free (newPtr);
17        return false;
18       } // if
19    else
20         return true;
21 }  // BST_Full
```

BST Count Application Interface

```
1  /* ================== BST_Count ==================
2     Returns number of nodes in tree.
3        Pre     tree has been created
4        Returns tree count
5  */
6  int BST_Count (BST_TREE* tree)
7  {
8  // Statements
9     return (tree->count);
10 }  // BST_Count
```

Destroy BST Application Interface

```
1  /* =============== BST_Destroy ==============
2     Deletes all data in tree and recycles memory.
3     The nodes are deleted by calling a recursive
4     function to traverse the tree in inorder sequence.
5        Pre      tree is a pointer to a valid tree
6        Post     All data and head structure deleted
7        Return   null head pointer
8  */
9  BST_TREE* BST_Destroy (BST_TREE* tree)
10 {
11 // Statements
12    if (tree)
13       _destroy (tree->root);
14
15    // All nodes deleted. Free structure
16    free (tree);
17    return NULL;
18 }  // BST_Destroy
```

Internal Destroy Function

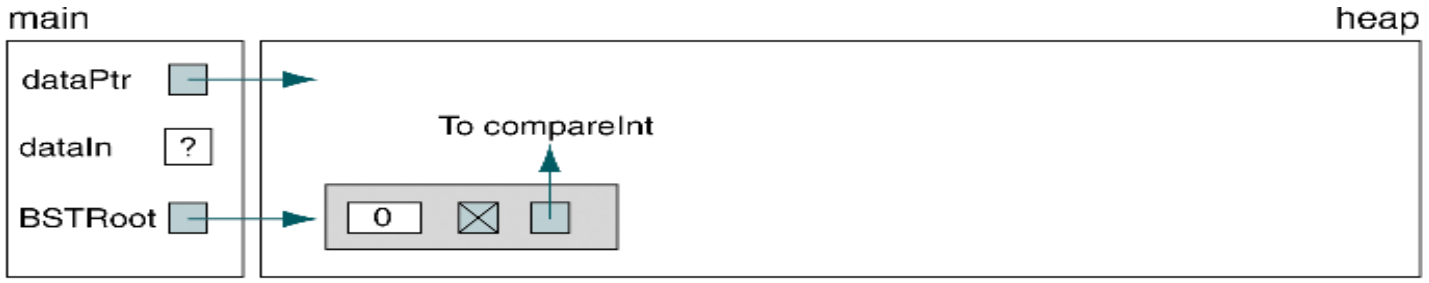
```
1  /* ============== _destroy ==============
2     Deletes all data in tree and recycles memory.
3     It also recycles memory for the key and data nodes.
4     The nodes are deleted by calling a recursive
5     function to traverse the tree in inorder sequence.
6        Pre      root is pointer to valid tree/subtree
7        Post     All data and head structure deleted
8        Return   null head pointer
9  */
10 void _destroy (NODE* root)
11 {
12 // Statements
13    if (root)
14       {
15        _destroy (root->left);
16        free (root->dataPtr);
17        _destroy (root->right);
18        free (root);
19       } // if
20    return;
21 } // _destroy
```

7-4 BST Applications

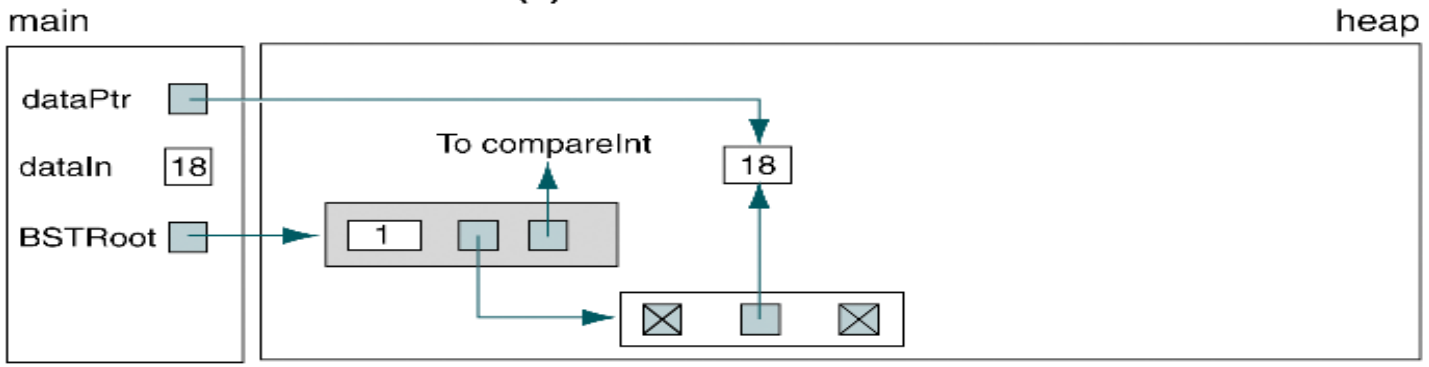
This section develops two applications that use the BST ADT. We begin the discussion of BST Applications with a simple application that manipulates integers. The second application, student list, requires a structure to hold the student's data.

- Integer Application
- Student List Application

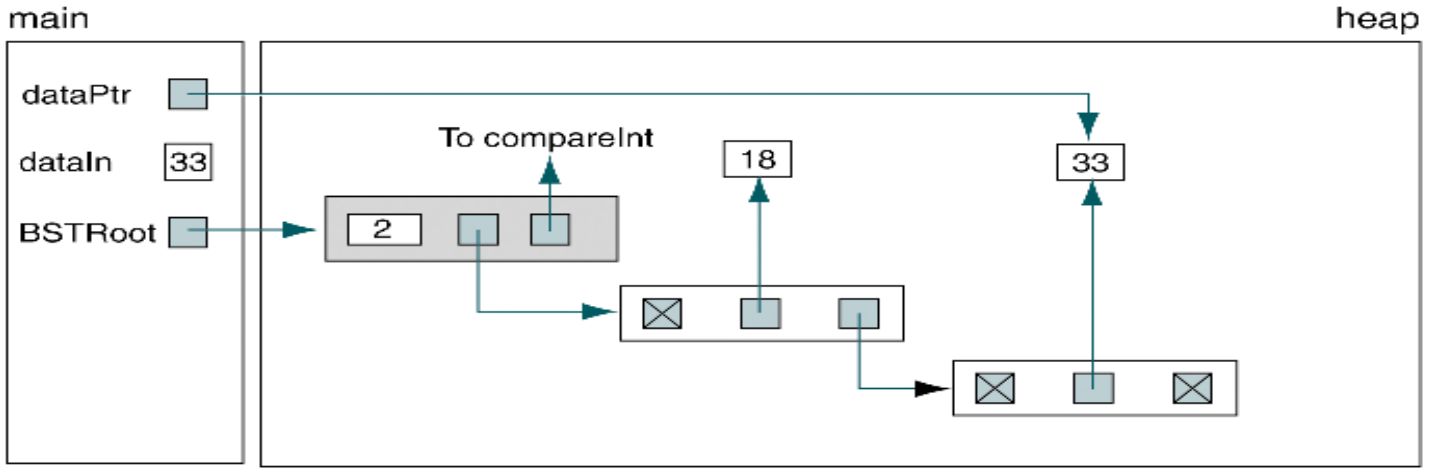
Insertions into a BST



(a) After tree creation



(b) After first insertion



(c) After second insertion

BST Integer Application

```
 1  /* This program builds and prints a BST.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7  #include "P7-BST-ADT.h"
 8
 9  // Prototype Declarations
10  int  compareInt (void* num1, void* num2);
11  void printBST   (void* num1);
12
13  int main (void)
14  {
15  // Local Definitions
16     BST_TREE* BSTRoot;
17     int*      dataPtr;
18     int       dataIn = +1;
19
```

BST Integer Application (cont.)

```
20  // Statements
21     printf("Begin BST Demonstation\n");
22
23     BSTRoot = BST_Create (compareInt);
24
25  // Build Tree
26     printf("Enter a list of positive integers;\n");
27     printf("Enter a negative number to stop.\n");
28
29     do
30         {
31          printf("Enter a number: ");
32          scanf ("%d", &dataIn);
33          if (dataIn > -1)
34              {
35               dataPtr = (int*) malloc (sizeof (int));
36               if (!dataPtr)
37                    {
38                     printf("Memory Overflow in add\n"),
39                     exit(100);
40                    } // if overflow
41               *dataPtr = dataIn;
42               BST_Insert (BSTRoot, dataPtr);
43              } // valid data
44         } while (dataIn > -1);
45
```

BST Integer Application (cont.)

```
46      printf("\nBST contains:\n");
47      BST_Traverse (BSTRoot, printBST);
48
49      printf("\nEnd BST Demonstration\n");
50      return 0;
51  }  // main
52
53  /* ==================== compareInt ====================
54      Compare two integers and return low, equal, high.
55          Pre  num1 and num2 are valid pointers to integers
56          Post return low (-1), equal (0), or high (+1)
57  */
58  int compareInt (void* num1, void* num2)
59  {
60  // Local Definitions
61      int key1;
62      int key2;
63
```

BST Integer Application (cont.)

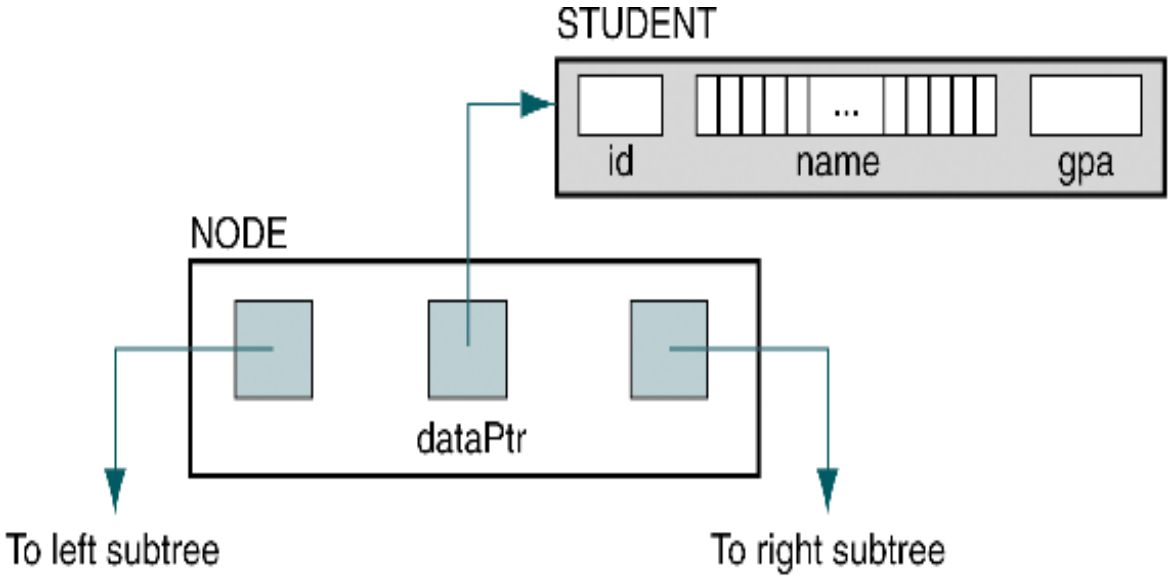
```
64  // Statements
65     key1 = *(int*)num1;
66     key2 = *(int*)num2;
67     if (key1 < key2)
68          return -1;
69     if (key1 == key2)
70          return 0;
71     return +1;
72  }  // compareInt
73
74  /* ==================== printBST ====================
       Print one integer from BST.
75          Pre  num1 is a pointer to an integer
76          Post integer printed and line advanced
77  */
78  void printBST (void* num1)
79  {
80  // Statements
81     printf("%4d\n", *(int*)num1);
82     return;
83  }  // printBST
84
```

```
Results:
Begin BST Demonstation
Enter a list of positive integers;
Enter a negative number to stop.
Enter a number: 18
```

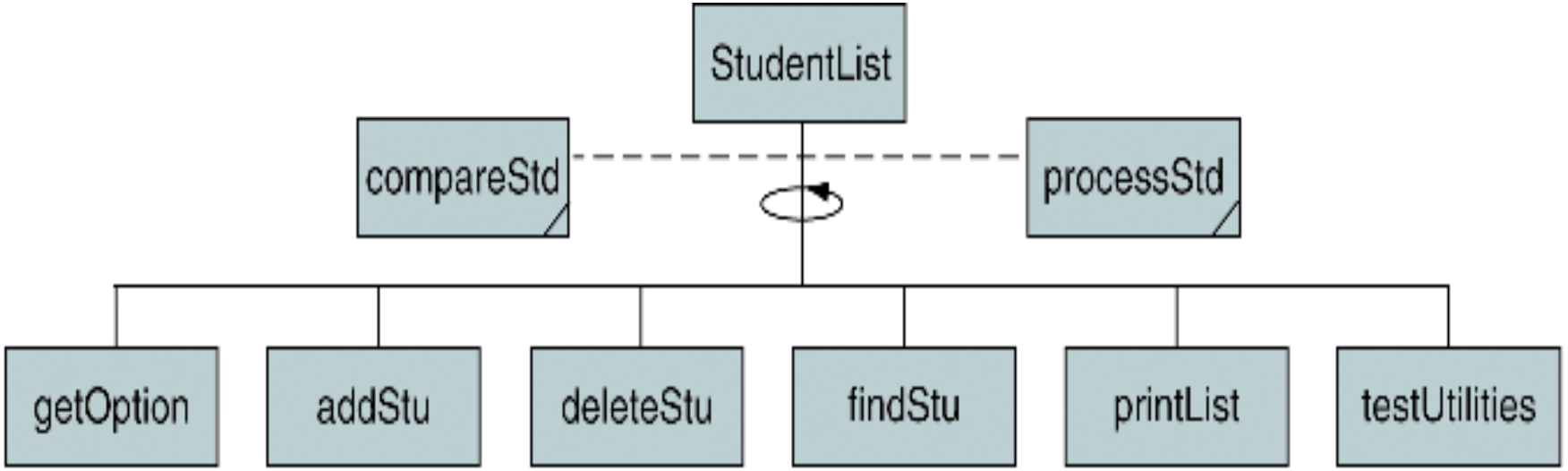
BST Integer Application (cont.)

```
Enter a number: 33
Enter a number: 7
Enter a number: 24
Enter a number: 19
Enter a number: -1

BST contains:
   7
  18
  19
  24
  33

End BST Demonstration
```

Student Data in ADT



Student List Design



BST Student Application

```
1  /* This program builds and prints a student list.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <ctype.h>
8
9  #include "P7-BST-ADT.h"
10
11 // Structures
12 typedef struct
13    {
14     int   id;
15     char  name[40];
16     float gpa;
```

BST Student Application (cont.)

```
17      } STUDENT;
18
19  // Prototype Declarations
20  char getOption    (void);
21  void addStu       (BST_TREE* list);
22  void deleteStu    (BST_TREE* list);
23  void findStu      (BST_TREE* list);
24  void printList    (BST_TREE* list);
25  void testUtilties (BST_TREE* tree);
26  int  compareStu   (void* stu1, void* stu2);
27  void processStu   (void* dataPtr);
28
29  int main (void)
30  {
31  // Local Definitions
32     BST_TREE* list;
33     char      option = ' ';
34
```

BST Student Application (cont.)

```
35  // Statements
36     printf("Begin Student List\n");
37     list = BST_Create (compareStu);
38
39     while ( (option = getOption ()) != 'Q')
40         {
41          switch (option)
42              {
43               case 'A': addStu (list);
44                         break;
45               case 'D': deleteStu (list);
46                         break;
47               case 'F': findStu (list);
48                         break;
49               case 'P': printList (list);
50                         break;
51               case 'U': testUtilties (list);
52                         break;
53             } // switch
54         } // while
55     list = BST_Destroy (list);
56
57     printf("\nEnd Student List\n");
58     return 0;
59  }  // main
60
61  /* ==================== getOption ====================
62     Reads and validates processing option from keyboard.
63          Pre  nothing
```

BST Student Application (cont.)

```
64          Post valid option returned
65  */
66  char getOption  (void)
67  {
68  // Local Definitions
69     char option;
70     bool error;
71
72  // Statements
73     printf("\n ======  MENU  ======\n");
74     printf(" A   Add Student\n");
75     printf(" D   Delete Student\n");
76     printf(" F   Find Student\n");
77     printf(" P   Print Class List\n");
78     printf(" U   Show Utilities\n");
79     printf(" Q   Quit\n");
80
```

BST Student Application (cont.)

```
81      do
82          {
83           printf("\nEnter Option: ");
84           scanf(" %c", &option);
85           option = toupper(option);
86           if   (option == 'A' || option == 'D'
87              || option == 'F' || option == 'P'
88              || option == 'U' || option == 'Q')
89                 error = false;
90           else
91                {
92                printf("Invalid option. Please re-enter: ");
93                 error = true;
94                } // if
95          } while (error == true);
96      return option;
97  }   // getOption
98
```

BST Student Application (cont.)

```
 99 /* ====================== addStu =======================
100    Adds a student to tree.
101        Pre  nothing
102        Post student added (abort on memory overflow)
103 */
104 void addStu (BST_TREE* list)
105 {
106 // Local Definitions
107    STUDENT* stuPtr;
108
109 // Statements
110    stuPtr = (STUDENT*)malloc (sizeof (STUDENT));
```

BST Student Application (cont.)

```
111     if (!stuPtr)
112         printf("Memory Overflow in add\n"), exit(101);
113
114     printf("Enter student id:   ");
115     scanf ("%d",  &(stuPtr->id));
116     printf("Enter student name: ");
117     scanf ("%39s", stuPtr->name);
118     printf("Enter student gpa:  ");
119     scanf ("%f",  &(stuPtr->gpa));
120
121     BST_Insert (list, stuPtr);
122 }   // addStu
123
124 /* ===================== deleteStu ====================
125     Deletes a student from the tree.
126         Pre  nothing
127         Post student deleted or error message printed
128 */
```

BST Student Application (cont.)

```
129 void deleteStu (BST_TREE* list)
130 {
131 // Local Definitions
132    int  id;
133    int* idPtr = &id;
134
135 // Statements
136    printf("Enter student id: ");
137    scanf ("%d", idPtr);
138
139    if (!BST_Delete (list, idPtr))
140       printf("ERROR: No Student: %0d\n", *idPtr);
141 }  // deleteStu
142
```

redundant

BST Student Application (cont.)

```
143 /* ==================== findStu ====================
144    Finds a student and prints name and gpa.
145        Pre  student id
146        Post student data printed or error message
147 */
148 void findStu (BST_TREE* list)
149 {
150 // Local Definitions
151    int      id;
152    STUDENT* stuPtr;
153
154 // Statements
155    printf("Enter student id: ");
156    scanf ("%d", &id);
157
```

BST Student Application (cont.)

```
158     stuPtr = (STUDENT*)BST_Retrieve (list, &id);
159     if (stuPtr)
160        {
161         printf("Student id:   %04d\n",  id);
162         printf("Student name: %s\n",    stuPtr->name);
163         printf("Student gpa:  %4.1f\n", stuPtr->gpa);
164        } // if
165     else
166        printf("Student %d not in file\n", id);
167  }  // findStu
168
```

BST Student Application (cont.)

```
169 /* ==================== printList ====================
170    Prints a list of students.
171        Pre  list has been created (may be null)
172        Post students printed
173 */
174 void printList  (BST_TREE* list)
175 {
176 // Statements
177    printf("\nStudent List:\n");
178    BST_Traverse (list, processStu);
179    printf("End of Student List\n");
180    return;
181 }  // printList
182
```

BST Student Application (cont.)

```
183 /* ================== testUtilties =================
184    This function tests the ADT utilities by calling
185    each in turn and printing the results.
186       Pre  tree has been created
187       Post results printed
188 */
189 void testUtilties (BST_TREE* tree)
190 {
191 // Statements
192    printf("Tree contains %3d nodes: \n",
193           BST_Count(tree));
194    if (BST_Empty(tree))
195        printf("The tree IS empty\n");
196    else
197        printf("The tree IS NOT empty\n");
198    if (BST_Full(tree))
199        printf("The tree IS full\a\n");
200    else
201        printf("The tree IS NOT full\n");
202    return;
203 }  // testUtilities
204
```

BST Student Application (cont.)

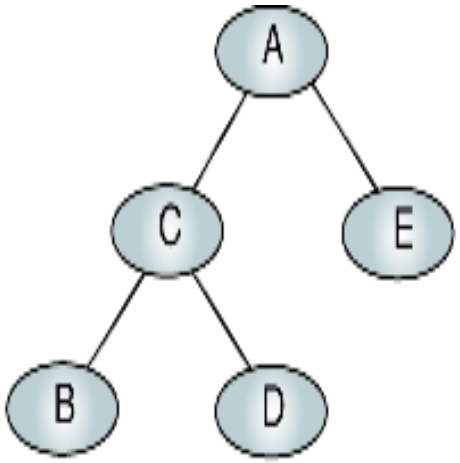
```
205 /* ==================== compareStu ====================
206    Compare two student id's and return low, equal, high.
207        Pre  stu1 and stu2 are valid pointers to students
208        Post return low (-1), equal (0), or high (+1)
209 */
210 int  compareStu (void* stu1, void* stu2)
211 {
212 // Local Definitions
213    STUDENT s1;
214    STUDENT s2;
215 
216 // Statements
217    s1 =  *(STUDENT*)stu1;
218    s2 =  *(STUDENT*)stu2;
219 
220    if ( s1.id < s2.id)
221          return -1;
222 
223    if ( s1.id == s2.id)
224          return 0;
225 
226    return +1;
227 }  // compareStu
228 
```

BST Student Application (cont.)

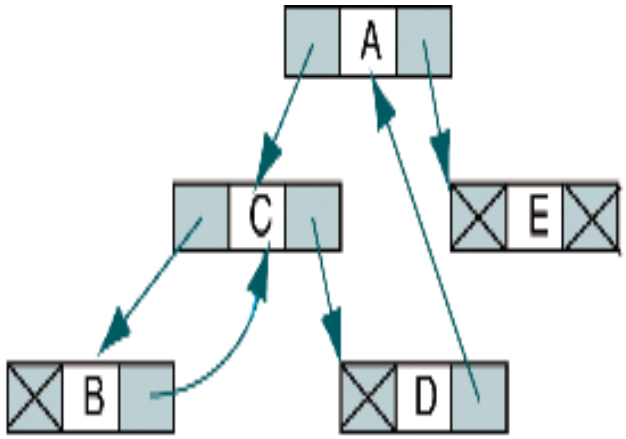
```
229 /* ================== processStu ====================
230    Print one student's data.
231        Pre  stu is a pointer to a student
232        Post data printed and line advanced
233 */
234 void processStu (void* stuPtr)
235 {
236 // Local Definitions
237    STUDENT aStu;
238
239 // Statements
240    aStu = *(STUDENT*)stuPtr;
241    printf("%04d  %-40s %4.1f\n",
242           aStu.id, aStu.name, aStu.gpa);
243    return;
244 }  // processStu
```

7-5 Threaded Trees 線索二元樹

- Recursion or stack are inefficient if the tree must be traversed frequently
- In a threaded tree, null pointers are replaced with pointers to their successor nodes
- The traversal is more efficient if the tree is a threaded tree



(a) Binary tree



(b) Threaded binary tree (inorder)