CS 2412 Data Structures

Chapter 5

Searching and Binary Search Trees

5.1 Searching sequence

The purpose of **searching**: find out some specific item from a collection of data.

Key of information: some kind of index of items in the collection of data.

External and internal searching: external searching used for search from databases. Here we basically consider algorithms for internal searching.

Key type usually are simple: usually `int, float, char *`

macro substitution with parameters:

macro substitution can adding parameters. Example:

```
#define GETBIT(w,n) (((unsigned int) (w) >> (n)) & 01)
```

The source code: `GETBIT(15,2)` will be replaces with `(((unsigned int) (15) >> (2)) & 01)`

Note: `>> n` moves right `n` bits.

The extra parentheses of `w, n` are necessary. Otherwise might cause problems if `w` is an expression.

In searching, we need to define equality and less than of keys.

```
#define EQ(a,b) ((a)==(b))

#define LT(a,b) ((a)<(b))
```

For characters:

```
#define EQ(a,b) (!strcmp((a),(b)))

#defing LT(a,b) (strcmp((a),(b))<0)
```

Sequential search

Consider sequential search in a list implemented by an array. The `ListEntry` contains a `KeyType key`.

```
int SequentialSearch(List list,KeyType target)
{
  int location;
  for(location=0;location<list.count;location++)
    if(EQ(list.entry[location].key,target))
      return location;
   return -1;
}
```

5.2 Complexity analysis and big-O notation

If an algorithm is linear, i.e., there is no loops, then its efficiency is a function of the number of instructions it contains. On the other hand, functions that use loops or recursion vary widely in efficiency. So we will focus on loops.

Linear loops examples:

```
for (i = 0; i< n; i++)
   application code

for (i = 0; i< n;  i += 2)
   application code
```

The complexity of above loops are proportional of n, denoted $O(n)$

Logarithmic loops:

```
for (i = 0; i< n; i *=2)
   application code
```

The complexity of the above loop is proportional of $\log_2 n$, denote $O(\log n)$

```
for (i = 0; i< n; i++)
   for( j = 1; j<=n; j*=2)
     application code
```

The complexity of the above nested loop is proportional of $n \log_2 n$, denoted $O(n \log n)$.

```
for (i = 0; i< n; i++)
   for (j = 1; j < n;  j++)
     application code
```

The complexity of the above nested loop is proportional of n^2, denoted $O(n^2)$.

Here we just gave some description of big-O notation, but not the formal definition.

Basic complexity: $O(\log n), O(n), O(n \log n), O(n^2), O(n^k), O(c^n)$, and $O(n!)$.

The complexity of sequential search

The most time consuming in the program is comparing. Suppose the size of the list is n. In average, how many compares need for the program?

$$\frac{1+2+3+\cdots+n}{n.}$$

This equals to:

$$\frac{n(n+1)}{2n} = \frac{1}{2}(n+1).$$

So it is proportional to n, $O(n)$.

Testing can also used to estimate CPU time.

In a test, a sample data is set up. Then run the program and count the CPU time used. We will assume the search is in random.

```
int RandomInt(int low, int high)
{
  if(low > high)
    Error("RandomInt: low cannot be greater than high.");
  return(high - low +1)*(rand()/(double)INT_MAX)+low;
}
```

```
#include<time.h>
float Time(int flag)
{
  static clock_t start;
  clock_t end;
  if(flag==START){
    start=clock();
    return 0.0;
  }else{
    end=clock();
    return (end-start)/CLK_TCK;
  }
}
```

```
void TestSearch(List list,int (*Search(List list,
       KeyType target), int searchcount)
{
  float elapsedtime;
  extern long compcounter; /*global comparison counter*/
  ......
  (void)Time(START);
  ......
    for(i=0;i<serchcount;i++){
     target=2*RandomInt(1,list.count)-1;
     if(Search(list,target)==-1)
       printf("%d not found\n",target);
    }
    ......
    elapsdtime=Time(END);
    ......
```

Binary sequential search

If the keys of a list are arranged in order, then we can use a very efficient search method: binary sequential search. In that case, only n times comparisons are needed for search a list of length 2^n.

Definition An *ordered list* is a list in which each entry contains a key, such that the keys are in order. That is, if entry i comes before entry j in the list, then the key of entry i is less than or equal to the key of entry j.

```
/*insert resulting an ordered list*/

void InsertOrder(List *list,ListEntry x)
{
  int current;
  ListEntry currententry;
  for(current=0;current<ListSize(*list);current++){
     currententry=RetrieveList(*list,current);
     if(LE(x.key,currententry.key))
     break;
  }
  InsertList(list,x,current);
}
```

In binary search algorithms, we use two indices `top` and `bottom`, such that the `target` is between the two indices. The key of `top` is greater than or equal to the key of index `bottom`.

Main idea: compute `middle=(top+bottom)/2` and compare the key at `middle` with `target`. Then move the `top` or `bottom` to `middle`. The binary search process terminates when `top <= bottom`.

The complexity of binary search for ordered sequence is $O(\log n)$.

Asymptotics

Asymptotics is an important concept to compare the efficiency of computer algorithms.

Definition If $f(n)$ and $g(n)$ are functions defined for positive integers, then to write

$$f(n) \text{ is } O(g(n))$$

means that there exists a constant c such that $|f(n)| \leq c|g(n)|$ for all sufficient large positive integers n.

Example.

If $f(n) = 4n + 200$, then $f(n)$ is $O(n)$.

If $f(n) = 0.001n^2$, then $f(n)$ is not $O(n)$, but $O(n^2)$.

If $f(n)$ is a polynomial in n with degree r, then $f(n)$ is $O(n^r)$, but is not $O(n^s)$ for any $s < r$.

Any logarithm of n grows more slowly (as n is increases) than any positive power of n. Hence $\log n$ is $O(n^k)$ for any $k > 0$, but n^k is never $O(\log n)$ for any power $k > 0$.

If $f(n) - g(n)$ is $O(h(n))$, then we define $f(n) = g(n) + O(h(n))$.

Running time for successful search a list of length n:

- Sequential search is $\frac{1}{2}n + O(1)$.
- Binary search is $2 \lg n + O(1)$.
- Retrieval from a contiguous list is $O(1)$.

Most common orders:
$O(1), O(\log n), O(n), O(n \log n), O(n^2), O(n^3), O(2^n)$

5.3 Introduction to trees

A tree consists of a finite set of elements, called nodes, and a finite set of directed lines, called branches, that connect the nodes.

NoteThe above definition is not a standard definition of trees. Usually, the above definition is called directed trees.

Some terminologies:

- nodes (vertices), branches (arcs, directed edges), degree, indegree, outdegree.
- root, leaf, internal node, parent, child, siblings, ancestor, descendant.
- path, level, height (depth), subtrees.

A recursive definition of a tree:

Definition A tree is a set of nodes that either:

1. Is empty, or
2. Has a designated node, called the root, from which hierarchically descend zero or more subtrees, which are also trees.

Binary tree is a tree in which no node can have more than two subtrees; the maximum outdegree for a node is two.

Properties:

- Height H of a binary trees of N nodes :
  $H_{max} = N, H_{min} = \lfloor \log_2 N \rfloor + 1$.
- Number of nodes N for a binary tree of height H:
  $N_{min} = H, N_{max} = 2^H - 1$.
- Balance factor of a binary tree: $B = H_L - H_R$, where H_L is the height of the left subtree and H_R is the height of the right subtree.

Some special binary trees:

- In a balanced binary tree, $-1 \leq B \leq 1$ for the tree and all the subtrees.
- A complete tree has the maximum number of entries for its height. (Th maximum number is reached when the last level is full).
- A tree is nearly complete if it has the minimum height for its nodes and all nodes in the last level are found on the left.

Binary tree Traversals

A binary tree traversal requires that each node of the tree be processed once and only once in a predetermined sequence.

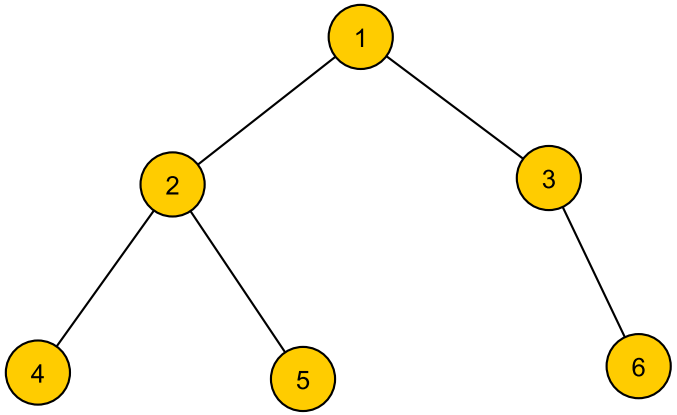
In a **breadth-first traversal**, the procession proceeds horizontally from the root to all of its children, then to its children's children, and so forth until all nodes have been processed. (Each level is completely processed before the next level is started).

Depth-first traversal

There are different orders for depth-first traversals. Let N, L, and R denote the root node, the left subtree and the right subtree, respectively. The following order is defined recursively.

- Preorder traversal (NLR)
- Inorder traversal (LNR)
- Postorder traversal (LRN)

Example



- Preorder (NLR): 1, 2, 4, 5, 3, 6.
- Inorder (LNR): 4, 2, 5, 1, 3, 6.
- Postorder (LRN): 4, 5, 2, 6, 3, 1.

Algorithms

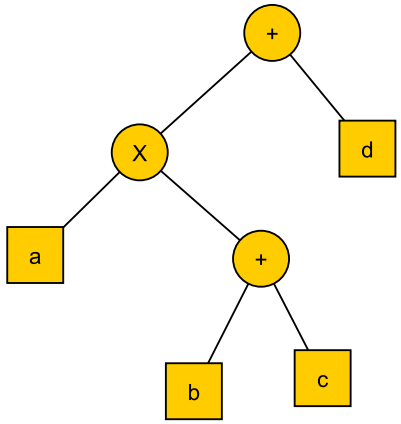
```
Algorithm preOrder (root)
 if (root is not null)
    process (root)
    preOrder(leftSubtree)
    preOrder(rightSubtree)
 end if
```

The algorithms for inorder and postorder are similar.

```
Algorithm breadthFirst (root)
  set currentNode to root
  createQueue (bfQueue)
  loop (currentNode not null)
    process (currentNode)
    if (left subtree not null)
      enqueue (bfQueue, left subtree)
    end if
    if (right subtree not unll)
      enqueue (bfQueue, right subtree)
    end if
    if (not emptyQueue(bfQueue))
     set currentNode to dequeue (bfQueue)
   else
     set currentNode to null
   end if
  end loop
  destroyQueue (bfQueue)
```

Example apply binary tree to arithmetic expression

An expression tree is a binary tree such that each leaf is an operand and root and internal nodes are operators.



$$a \times (b + c) + d$$

For an expression tree, the three depth-first traversals represent the three different expression: infix , postfix, prefix expressions.

```
Algorithm infix (tree)
  if (tree not empty)
    if (tree-token is an operand)
     print (tree-token)
   else
     print (open parenthesis)
     infix(left subtree)
     print(tree-token)
     infix(right subtree)
     print (close parenthesis)
  end if
```

For postfix and prefix, we don't need to add parentheses to the expression.

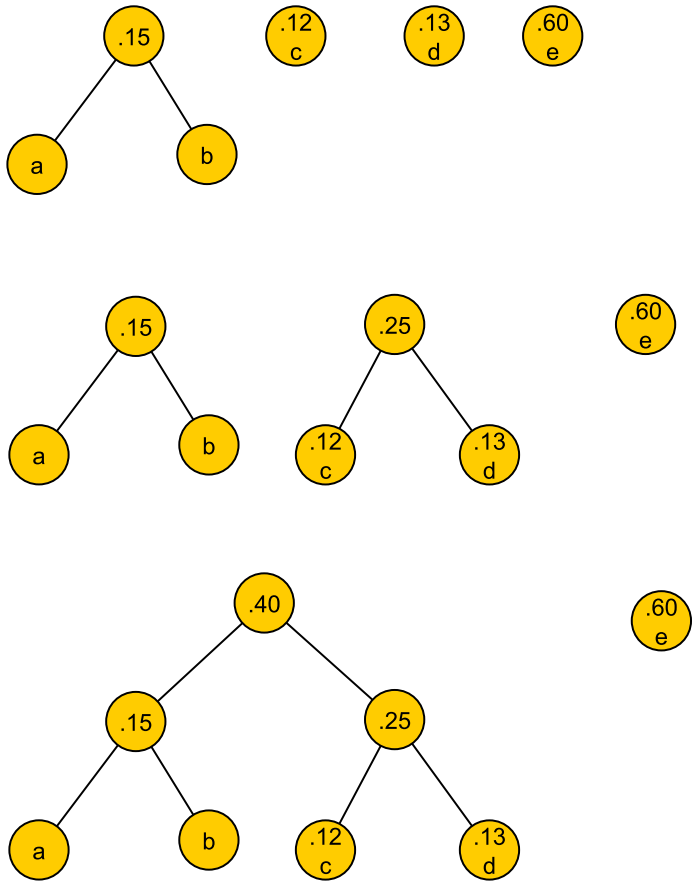
```
Algorithm postfix (tree)
  if (tree not empty)
    postfix (left subtree)
    postfix(right subtree)
    print (tree-token)
  end if
```

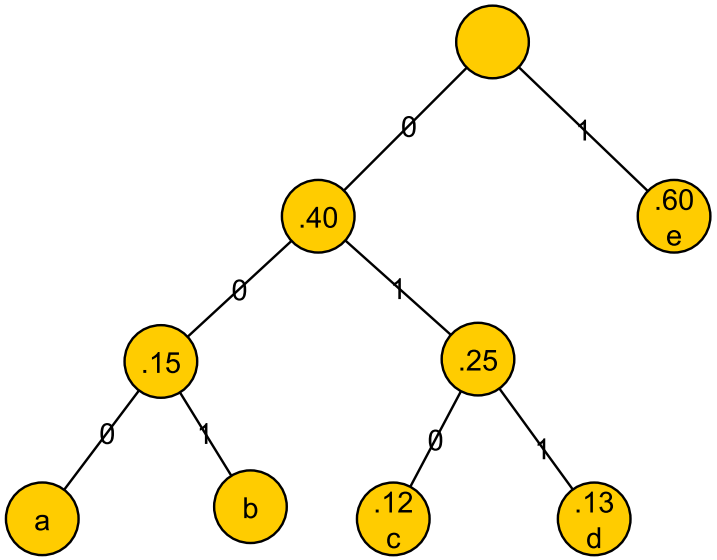
The `Algorithm prefix` is similar.

Example Huffman code (Huffman encoding)

Huffman encoding encodes elements of a set X into binary codes, such that the length of the resulting codes is optimal. So more frequent elements will encoding to shorter codes. In each iteration, the two elements having lowest probability are combined into one element as their parent with the probability the sum of the two smaller probability. When only one element remains, the constructed binary tree becomes the Huffman tree. And this tree can be used for encoding.

Suppose $X = \{a, b, c, d, e\}$. The probability for them are $.05, .10, .12, .13, .60$. The Huffman tree can be formed as follows.





So the code will be:

$a : 000, b : 001, c : 010, d : 011, e : 1$

5.4 Binary search trees

A binary search tree (BST) is a binary tree with the following properties:

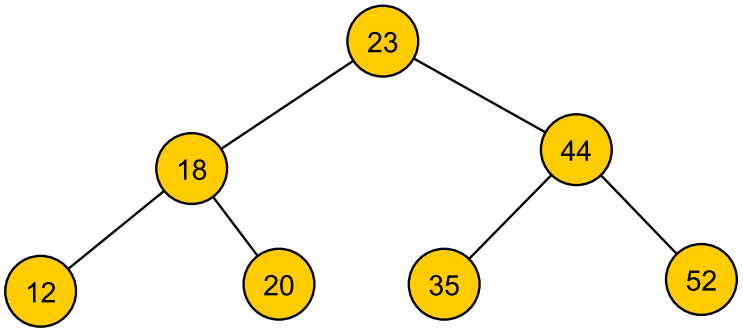
- All items in the left subtree are less than the root,
- All items in the right subtree are greater than or equal to the root.
- Each subtree is itself a binary search tree.

In general cases, " item" means the "key of the node".

BST operations

- Traversals: inorder, preorder, postorder.
- Searches: smallest, largest, others.
- Insertion
- Deletion

Traversal:



- Preorder: 23, 18, 12, 20, 44, 35, 52.
- Postorder: 12, 20, 18, 35, 52, 44, 23.
- Inorder: 12, 18, 20, 23, 35, 44, 52.

Search:

- Find smallest:

```
Algorithm findSmallestBST (tree)
    if (left subtree empty)
      return root
    end if
    return findSmallestBST(left subtree)
```

- Find largest:

```
Algorithm findLargestBST (tree)
    if (right subtree empty)
      return root
    end if
    return findLargestBST(right subtree)
```

- Algorithm searchBST (tree, targetKey)

```
if (empty tree)
  return null //Not found
end if
if (targetKey < root )
  return searchBST (left subtree, targetKey)
else if (targetKey > root )
  return searchBST (right subtree, targetKey)
else
  return root  //found the key
end if
```

The above algorithms used recursive methods. It is not difficult to write non-recursive algorithms.

Insertion

```
Algorithm addBST (root, newNode)
 if (empty tree)
   set root to newNode
   return newNode
 end if
 if (newNode < root)
   return addBST (left subtree, newNode)
 else
  return addBST (right subtree, newNode)
 end if
```

Deletion

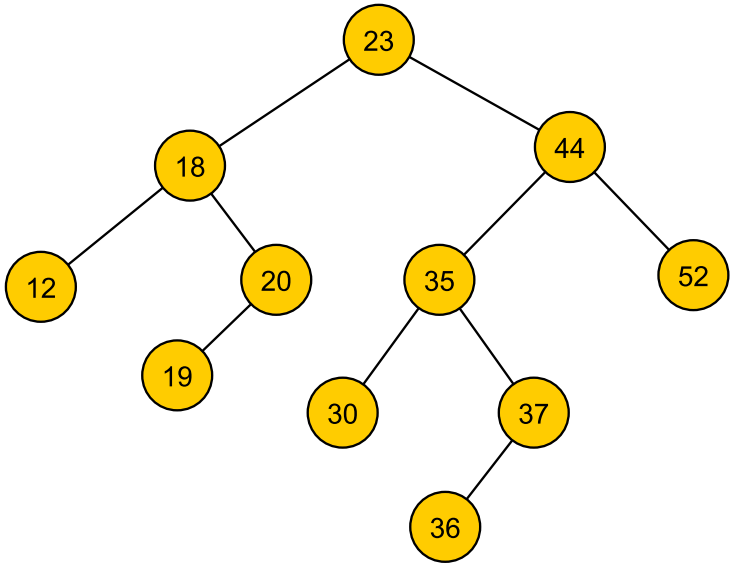
```
Algorithm deleteBST (root, dltKey)
  if (empty tree)
    return false
  end if
  if (dltKey < root)
    return deleteBST (left subtree, dltKey)
  else if (dltKay > root)
    return deleteBST (right subtree, dltKey)
  else   //found the node
     if (no left subtree)
       make right subtree the root
     else if (no right subtree)
       make left subtree the root
     else
       save root in deleteNode
```

```
    set largest to largestBST(left subtree)
    move data in largest to deleteNode
    return deleteBST (left subtree, key of largest)
  end if
```

Note

When the deleted node has both right subtree and left subtree, we can move the smallest of the right subtree to the deleted node instead of moving the largest of the left subtree to the deleted node.

Example: Delete 12, 18, 44 from the following BST.



BST ADT program

```
#include <stdbool.h>

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

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

```
// Prototype Declarations
BST_TREE* BST_Create
          (int (*compare) (void* argu1, void* argu2));
BST_TREE* BST_Destroy (BST_TREE* tree);

bool  BST_Insert   (BST_TREE* tree, void* dataPtr);
bool  BST_Delete   (BST_TREE* tree, void* dltKey);
void* BST_Retrieve (BST_TREE* tree, void* keyPtr);
void  BST_Traverse (BST_TREE* tree,
                    void (*process)(void* dataPtr));

bool BST_Empty (BST_TREE* tree);
bool BST_Full  (BST_TREE* tree);
int  BST_Count (BST_TREE* tree);
```

```
static NODE* _insert
               (BST_TREE* tree, NODE* root,
                NODE* newPtr);
static NODE* _delete
               (BST_TREE* tree,    NODE* root,
                void*     dataPtr, bool* success);
static void* _retrieve
               (BST_TREE* tree,
              void* dataPtr, NODE* root);
static void _traverse
               (NODE* root,
                void (*process) (void* dataPtr));
static void _destroy (NODE* root);
```

```
BST_TREE* BST_Create
        (int  (*compare) (void* argu1, void* argu2))
{
BST_TREE* tree;

tree = (BST_TREE*) malloc (sizeof (BST_TREE));
if (tree)
   {
    tree->root    = NULL;
    tree->count   = 0;
    tree->compare = compare;
   } // if

return tree;
} // BST_Create
```

```
bool BST_Insert (BST_TREE* tree, void* dataPtr)
{
NODE* newPtr;
newPtr = (NODE*)malloc(sizeof(NODE));
if (!newPtr)
   return false;
newPtr->right   = NULL;
newPtr->left    = NULL;
newPtr->dataPtr = dataPtr;
if (tree->count == 0)
    tree->root  =  newPtr;
else
    _insert(tree, tree->root, newPtr);
(tree->count)++;
return true;
}  // BST_Insert
```

```
NODE* _insert (BST_TREE* tree, NODE* root, NODE* newPtr)
{
if (!root)
   return newPtr;
if (tree->compare(newPtr->dataPtr,
                  root->dataPtr) < 0)
   {
    root->left = _insert(tree, root->left, newPtr);
    return root;
   } // new < node
else
   {
    root->right = _insert(tree, root->right, newPtr);
    return root;
     } // else new data >= root data
return root;
} // _insert
```

```
bool BST_Delete (BST_TREE* tree, void* dltKey)
{
bool  success;
NODE* newRoot;

newRoot = _delete (tree, tree->root, dltKey, &success);
if (success)
   {
    tree->root = newRoot;
    (tree->count)--;
    if (tree->count == 0)
        tree->root = NULL;
   } // if
return success;
}  // BST_Delete
```

```
NODE*  _delete (BST_TREE* tree,    NODE* root,
                void*     dataPtr, bool* success)
{
// Local Definitions
NODE* dltPtr;
NODE* exchPtr;
NODE* newRoot;
void* holdPtr;

// Statements
if (!root)
   {
    *success = false;
    return NULL;
   } // if
```

```
if (tree->compare(dataPtr, root->dataPtr) < 0)
     root->left  = _delete (tree,    root->left,
                            dataPtr, success);
else if (tree->compare(dataPtr, root->dataPtr) > 0)
     root->right = _delete (tree,    root->right,
                            dataPtr, success);
else
    // Delete node found--test for leaf node
    {
     dltPtr = root;
     if (!root->left)
         // No left subtree
         {
          free (root->dataPtr);        // data memory
          newRoot = root->right;
          free (dltPtr);               // BST Node
```

```
        *success = true;
        return newRoot;                 // base case
      } // if true
  else
      if (!root->right)
          // Only left subtree
          {
           newRoot = root->left;
           free (dltPtr);
           *success = true;
           return newRoot;              // base case
         } // if
      else
          // Delete Node has two subtrees
          {
           exchPtr = root->left;
```

```
            // Find largest node on left subtree
            while (exchPtr->right)
                exchPtr = exchPtr->right;

            // Exchange Data
            holdPtr           = root->dataPtr;
            root->dataPtr     = exchPtr->dataPtr;
            exchPtr->dataPtr  = holdPtr;
            root->left        =
               _delete (tree,   root->left,
                        exchPtr->dataPtr, success);
          } // else
    } // node found
return root;
} // _delete
```

```
void* BST_Retrieve  (BST_TREE* tree, void* keyPtr)
{
// Statements
if (tree->root)
    return _retrieve (tree, keyPtr, tree->root);
else
    return NULL;
} // BST_Retrieve
```

```
void* _retrieve (BST_TREE* tree,
                 void* dataPtr, NODE* root)
{
if (root)
    {
     if (tree->compare(dataPtr, root->dataPtr) < 0)
         return _retrieve(tree, dataPtr, root->left);
     else if (tree->compare(dataPtr, root->dataPtr) > 0)
         return _retrieve(tree, dataPtr, root->right);
     else
         // Found equal key
         return root->dataPtr;
    }  // if root
else
    return NULL;
} // _retrieve
```

```
void BST_Traverse (BST_TREE* tree,
                   void (*process) (void* dataPtr))
{
// Statements
_traverse (tree->root, process);
return;
}  // end BST_Traverse
```

```
void _traverse (NODE* root,
                void (*process) (void* dataPtr))
{
// Statements
if  (root)
    {
     _traverse (root->left, process);
     process   (root->dataPtr);
     _traverse (root->right, process);
    } // if
return;
}  // _traverse
```

Example Build and print a BST of integers

```
#include <stdio.h>
#include <stdlib.h>
#include "P7-BST-ADT.h"
int  compareInt (void* num1, void* num2);
void printBST   (void* num1);

int main (void)
{
BST_TREE* BSTRoot;
int*      dataPtr;
int       dataIn = +1;
printf("Begin BST Demonstation\n");
BSTRoot = BST_Create (compareInt);
printf("Enter a list of positive integers;\n");
printf("Enter a negative number to stop.\n");
```

```
do
   {
    printf("Enter a number: ");
    scanf ("%d", &dataIn);
    if (dataIn > -1)
       {
        dataPtr = (int*) malloc (sizeof (int));
        if (!dataPtr)
            {
             printf("Memory Overflow in add\n"),
             exit(100);
            } // if overflow
        *dataPtr = dataIn;
        BST_Insert (BSTRoot, dataPtr);
       } // valid data
   } while (dataIn > -1);
```

```
printf("\nBST contains:\n");
BST_Traverse (BSTRoot, printBST);

printf("\nEnd BST Demonstration\n");
return 0;
} // main
```

The program uses ADT BST. So `void *` are used, which need to cast when used.

In ADT BST, the create function needs a compare function and the traverse needs a process function. So `comareInt` and `printBST` are implemented as follows.

```
int compareInt (void* num1, void* num2)
{
int key1;
int key2;

key1 = *(int*)num1;
key2 = *(int*)num2;
if (key1 < key2)
    return -1;
if (key1 == key2)
    return 0;
return +1;
} // compareInt
```

```
void printBST (void* num1)
{
// Statements
printf("%4d\n", *(int*)num1);
return;
} // printBST
```

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

An example of application of BST

This example creates a student list. It requires three pieces of data: the student's ID, the student's name, and the student's grade-point average. Students are added an deleted from the keyboard. They can be retrieved individually or as a list.

Some C codes are displayed below. The details are in the textbook.

```
typedef struct
   {
    int   id;
    char  name[40];
    float gpa;
   } STUDENT;

char getOption    (void);
void addStu       (BST_TREE* list);
void deleteStu    (BST_TREE* list);
void findStu      (BST_TREE* list);
void printList    (BST_TREE* list);
void testUtilties (BST_TREE* tree);
int  compareStu   (void* stu1, void* stu2);
void processStu   (void* dataPtr);
```

```
int main (void)
{
BST_TREE* list;
char      option = ' ';
printf("Begin Student List\n");
list = BST_Create (compareStu);

while ( (option = getOption ()) != 'Q')
   {
    switch (option)
       {
        case 'A': addStu (list);
                  break;
        case 'D': deleteStu (list);
                  break;
        case 'F': findStu (list);
```

```
                    break;
          case 'P': printList (list);
                    break;
          case 'U': testUtilties (list);
                    break;
       } // switch
    } // while
list = BST_Destroy (list);

printf("\nEnd Student List\n");
return 0;
} // main
```

```
char getOption  (void)
{
char option;
bool error;
printf("\n ======  MENU  ======\n");
printf(" A   Add Student\n");
printf(" D   Delete Student\n");
printf(" F   Find Student\n");
printf(" P   Print Class List\n");
printf(" U   Show Utilities\n");
printf(" Q   Quit\n");
do
   {
    printf("\nEnter Option: ");
    scanf(" %c", &option);
    option = toupper(option);
```

```
    if   (option == 'A' || option == 'D'
       || option == 'F' || option == 'P'
       || option == 'U' || option == 'Q')
          error = false;
    else
         {
          printf("Invalid option. Please re-enter: ");
          error = true;
         } // if
   } while (error == true);
return option;
} // getOption
```

```
 int  compareStu (void* stu1, void* stu2)
{
STUDENT s1;
STUDENT s2;
s1 =  *(STUDENT*)stu1;
s2 =  *(STUDENT*)stu2;

if ( s1.id < s2.id)
      return -1;

if ( s1.id == s2.id)
      return 0;

return +1;
} // compareStu
```