5 Binary Search Trees

5.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing *binary search trees* (BSTs).
- To get hands-on experience with implementing BSTs, using dynamically allocated memory.
- To develop critical thinking concerning implementation decisions for trees.

The outcomes for this session are:

- Improved skills in C programming using pointers and dynamic memory allocation.
- A clear understanding of BSTs and the operations which they support.
- The ability to make implementation decisions concerning trees.
- The ability to decide when BSTs are suitable in solving a problem.

5.2 Brief Theory Reminder

Binary search trees (BST), also called *ordered* or *sorted binary trees*, are data structures that store "items" (such as numbers, names etc.) in memory. BSTs allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its key (e.g., finding the name of a person by his/hers social security number).

Binary search trees keep their keys in sorted order. Thus, lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree. Thus, each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on *hash tables*.

BSTs are used quite frequently for fast storing and retrieval of information, based on keys. The keys are stored in the nodes, and must be distinct. Figure 5.1 shows an example of a BST which stores keys 2, 5, 6, 9, 10, 13, 14.

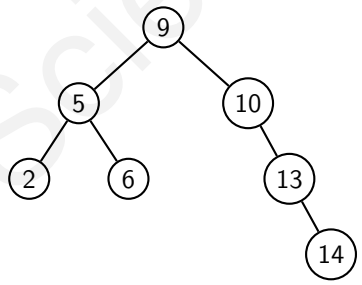


Figure 5.1: An example of a binary search tree

Binary Search Tree Structure

A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right. The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree.

A structure for a BST node may be implemented as shown by Listing 5.1.

Listing 5.1: Example binary tree node structure

```
typedef struct node_type
{
  KeyType key;
  ElementType info;
  struct node_type *left, *right;
} BSTNodeT;
```

The root of the tree may then be held in a variable declared like `BSTNodeT *root;`

The resulting structure of a BST depends upon the order of node insertions.

Inserting a Node in a BST

A BST is built by inserting new nodes into it. This can be done by performing the following steps:

1. If the tree is empty, then create a new node, the root, with key value *key*, and both left and right subtrees empty.
2. If root key is *key*, then insertion is not possible, as the keys must be distinct – nothing to do.
3. If the given key, say *key*, is less than the key stored at the root node, then continue with the first step, taking the left child as root.
4. If the given key, say *key*, is bigger than the key stored at the root node, then continue with the first step, taking the right child as root node.

The new node is attached on either left or right of an existing node.

A sequence of insertions in a BST is illustrated by Figure 5.2.

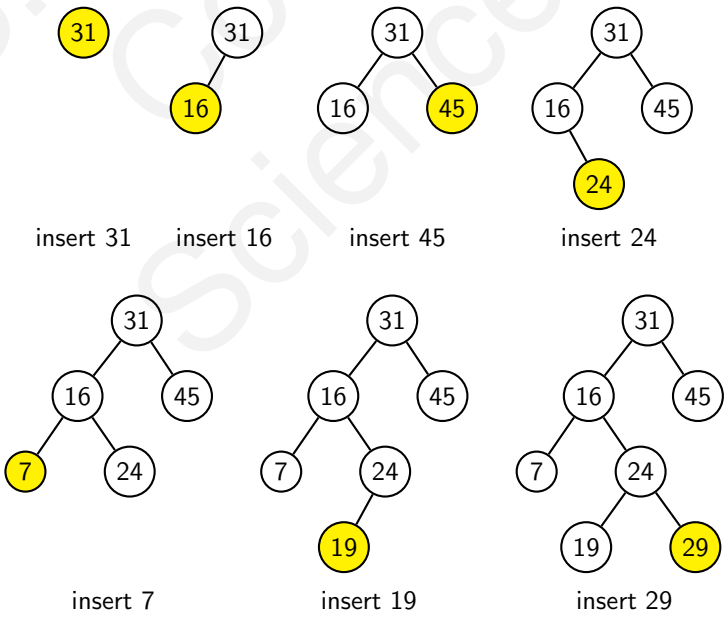


Figure 5.2: A sequence of insertion operations in a BST

Insertion can be achieved non-recursively using the procedure presented in Listing 5.2 or recursively as shown in Listing 5.3.

Listing 5.2: A non-recursive BST insertion procedure

```
typedef int KeyType;
typedef int ElementType;
/* for simplicity. Both should occur before BSTNodeT declaration */
BSTNodeT *createNode(KeyType givenKey)
{
    /* build node */
    BSTNodeT *p = (BSTNodeT *) malloc (sizeof (BSTNodeT));
    if (p)
    {
        p->key = givenKey;
        /* the info field should be filled in here */
        p->left = p-> right = NULL; /* leaf node */
    }
    return p;
}
void nInsert(BSTNodeT **root, KeyType givenKey)
{
    BSTNodeT *p = createNode(givenKey);
    if (!p)
    {
        // add suitable error message and give up
        abort();
    }
    if (*root == NULL)
    {
        /* empty tree */
        *root = p;
        return;
    }
    /* if we reach here then the tree is not empty;
       look for parent node of node pointed to by p */
    BSTNodeT *q = *root;
    for (; ;)
    {
        if (givenKey < q->key)
        {
            /* follow on left subtree */
            if (q-> left == NULL)
            {
                /* insert node here */
                q->left = p;
                return;
            }
            /* else */
            q = q->left;
        }
        else if (givenKey > q->key)
        {
            /* follow on right subtree */
            if (q-> right == NULL)
            {
                /* insert node here */
                q->right = p;
                return;
            }
            /* else */
            q = q->right;
        }
        else
        {
            /* key already present; could write a message... */
            free(p);
        }
    }
}
```

Listing 5.3: A recursive BST insertion procedure

```
// uses the createNode function shown before
BSTNodeT *insertNode(BSTNodeT *root, int key)
{
    /* If the tree is empty, then a single node tree is created */
    if (root == NULL)
        root = createNode(key);
    else
    {
        if (key < root->key)
            root->left = insertNode(root->left, key);
        else if (key > root->key)
            root->right = insertNode(root ->right, key);
        else
            printf("\nNode with key = %d already exists\n", key);
    }
    return root;
}
```

Finding a Node of a Given Key in a BST

The algorithm for finding a node is quite similar to the one for insertion. One important aspect when looking for information in a BST node is the amount of time taken. The number of comparisons would be optimal if there would be at most $\lceil \log_2 n \rceil$ where n is the number of nodes stored. The worst case is encountered when insertion is executed with the nodes sorted in either ascending or descending order. In that case the tree turn out to be a list.
A *find* function may be implemented as[1] shown by Listing 5.4 Invoke with, e.g. `q = find(root, key);`. Listing 5.5 shows how to find a node with minimum or maximum key value.

Listing 5.4: Finding a node with a given key in a BST

```
BStNodeT *find(BSTNodeT *root, KeyType givenKey)
{
  if (root == NULL) return NULL; /* empty tree */
  for (BSTNodeT *p = root; p != NULL;)
  {
    if (givenKey == p->key) return p;
    else
    if (givenKey < p->key) p = p->left;
    else                   p = p->right;
  }
  return NULL; /* not found */
}
```

Listing 5.5: Finding a node of minimum/maximum key value in a BST

```
BSTNodeT *findMin(BSTNodeT *node)
{
    if(node == NULL)
    {   /* There is no element in the tree */
        return NULL;
    }
    if(node->left) /* Go to the left sub tree to find the min element */
        return findMin(node->left);
    else
        return node;
}
BSTNodeT *findMax(BSTNodeT *node)
{
    if(node == NULL)
    {   /* There is no element in the tree */
        return NULL;
    }
    if(node->right) /* Go to the left sub tree to find the min element */
        return findMax(node->right);
    else
        return node;
}
```

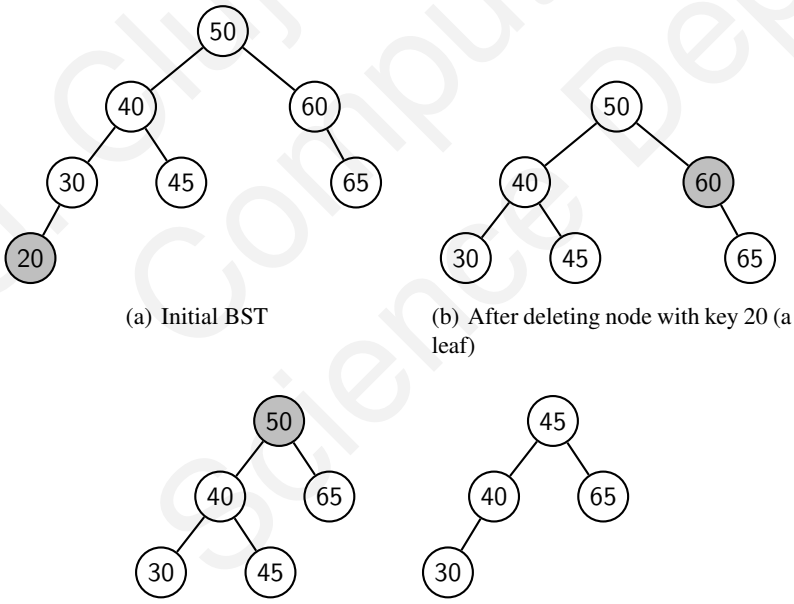
[1] see remarks on KeyType from previous section

Deleting a Node in a BST

When a node is deleted, the following situations are met:

1. The node to delete is a *leaf*. Child node pointer in parent node (left or right) must be set to zero (NULL).
2. The node to delete has only one child. In this case, in parent node of the one to be deleted its address is replaced by the address of its child.
3. The node to be deleted has two descendants. Replace the node to be deleted with either the leftmost node of right subtree or with the rightmost node of left subtree.

When deleting, we first have to look for the node to delete (based on some *key*) and then evaluate the situation according to the previous discussion. Figure 5.3 illustrates the deletion operation cases.



(a) Initial BST

(b) After deleting node with key 20 (a leaf)

(c) After deleting node with key 60 (that node had a single child with key 65)

(d) After deleting node with key 50 (that node had two children, with keys 40 and 60, respectively; find the inorder successor and replace the deleted node with it)

Figure 5.3: A sequence of node deletions in a BST. The node to be deleted is colored gray.

Listing 5.6 shows and implementation of a delete node function.

Listing 5.6: Deleting a node with a given key in a BST

```
BSTNodeT* delNode(BSTNodeT* node, int key)
{
    if(node == NULL)
    {
        printf("Element Not Found");
    }
    else if(key < node->key)
    {
        // must be in left subtree
        node->left = delNode(node->left, key);
    }
    else if(key > node->key)
    {
        // must be in right subtree
        node->right = delNode(node->right, key);
    }
    else
    {
        // found node
        /* Now We can delete this node and replace with either minimum element
           in the right sub tree or maximum element in the left subtree */
        if(node->right && node->left)
        {
            /* Here we will replace with minimum element in the right sub tree */
            BSTNodeT *temp = findMin(node->right);
            node -> key = temp->key;
            /* As we replaced it with some other node, we have to delete that node */
            node -> right = delNode(node->right, temp->key);
        }
        else
        {
            /* If there is only one or zero children then we can directly
               remove it from the tree and connect its parent to its child */
            BSTNodeT *temp = node;
            if(node->left == NULL)
                node = node->right;
            else if(node->right == NULL)
                node = node->left;
            free(temp); /* temp is longer required */
        }
    }
    return node;
}
```

Deleting a Whole BST

When total removal of the nodes of a BST is required, that can be accomplished by a postorder traversal of the tree and node-by-node removal using a recursive function similar to the one shown in Listing 5.7.

Listing 5.7: Deleting a BST

```
void delTree(BSTNodeT *root)
{
  if (root != NULL)
  {
    delTree(root ->left);
    delTree(root ->right);
    free(root);
  }
}
```

BST Traversals

As with any tree, there are three systematic traversals: preorder, inorder and postorder traversals. Listing 5.8 shows code implementing traversals.

Listing 5.8: Traversals of a BST

```
void preorder(BSTNodeT *p)
{
  if (p != NULL)
  {
    /* include code for info retrieval here */
    preorder(p->left);
    preorder(p->right);
  }
}
void inorder(BSTNOdeT *p)
{
  if (p != NULL)
  {
    inorder(p->left);
    /* include code for info retrieval here */
    inorder(p->right);
  }
}
void postorder(BSTNOdeT *p)
{
  if (p != NULL)
  {
    postorder(p->left);
    postorder(p->right);
    /* include code for info retrieval here */
  }
}
```

5.3 A Complete Implementation

Listing 5.9: Binary search trees code sample.

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

#define LEFT 1
#define RIGHT 2

typedef struct bst_node
{
  int key;
  struct bst_node *left, *right;
} BSTNodeT;

BSTNodeT *root;

void fatalError(const char *msg)
{
 printf(msg);
 printf("\n");
 exit (1);
}

/* Tree traversal functions
 * p = current node;
 * level = used for nice printing
*/
void preorder(BSTNodeT *p, int level)
{
  int i;

  if (p != NULL)
  {
    for (i = 0; i <= level; i++) printf(" "); /* for nice listing */
    printf("%2.2d\n", p->key);
    preorder(p->left, level + 1);
    preorder(p->right, level + 1);
  }
}
```

```
void inorder(BSTNodeT *p, int level)
{
  int i;

  if (p != NULL)
  {
    inorder(p->left, level + 1);
    for (i = 0; i <= level; i++) printf(" "); /* for nice listing */
    printf("%2.2d\n", p->key);
    inorder(p->right, level + 1);
  }
}
void postorder(BSTNodeT *p, int level)
{
  int i;

  if (p != NULL)
  {
    postorder(p->left, level + 1);
    postorder(p->right, level + 1);
    for (i = 0; i <= level; i++) printf(" "); /* for nice listing */
    printf("%2.2d\n", p->key);
  }
}
/* non recursive version of insert */
void insert(int key)
{
  BSTNodeT *p, *q;

  p = (BSTNodeT *) malloc(sizeof(BSTNodeT));
  p->key = key;
  p->left = p->right = NULL;
  if (root == NULL)
  {
    root = p;
    return;
  }
  q = root;
  for (; ;)
  {
    if (key < q->key)
    {
      if (q->left == NULL)
      {
        q->left = p;
        return;
      }
      else
        q = q->left;
    }
    else
    if (key > q->key)
    {
      if (q->right == NULL)
      {
        q->right = p;
        return;
      }
      else
            q = q->right;
    }
    else
    { /* keys are equal */
      printf("\nNode of key=%d already exists\n",  key);
      free(p);
      return;
    }
  }
}
BSTNodeT *recInsert(BSTNodeT *root, int key)
{
  BSTNodeT *p;

  if (root == NULL)
```

```
  {
    p = (BSTNodeT *) malloc(sizeof(BSTNodeT));
    p->key = key;
    p->left = p->right = NULL;
    root = p;
    return root;
  }
  else
  {
    if (key < root->key)
      root->left = recInsert(root ->left, key);
    else
    if (key > root->key)
      root->right = recInsert(root ->right, key);
    else /* key already there */
      printf("\nNode of key=%d already exists\n",
              key);
  }
  return root;
}
BSTNodeT *find(BSTNodeT *root, int key)
{
  BSTNodeT *p;

  if (root == NULL) return NULL;
  p = root;
  while (p != NULL)
  {
    if (p -> key == key)
      return p; /* found */
    else
    if (key < p->key)
          p = p->left;
    else
          p = p->right;
  }
  return NULL; /* not found */
}
BSTNodeT *delNode(BSTNodeT *root, int key)
{
  BSTNodeT *p; /* points to node to delete */
  BSTNodeT *pParent; /* points to parent of p */
  BSTNodeT *pRepl; /* points to node that will replace p */
  BSTNodeT *pReplParent; /* points to parent of node that will replace p */
  int direction; /* LEFT, RIGHT */
  if (root == NULL) return NULL; /* empty tree */
  p = root;
  pParent = NULL;
  while (p != NULL && p->key != key)
  {
    if (key < p->key)
    { /* search left branch */
      pParent = p;
      p = p->left;
      direction = LEFT;
    }
    else
    { /* search right branch */
      pParent = p;
      p = p->right;
      direction = RIGHT;
    }
  }
  if (p == NULL)
  {
    printf("\nNo node of key value=%d\n", key);
    return root;
  }
  /* node of key p found */
  if (p->left == NULL)
    pRepl = p->right; /* no left child */
  else
  if (p->right == NULL)
    pRepl = p->left; /* no right child */
```

```
  else
  { /* both children present */
    pReplParent = p;
    pRepl = p->right; /* search right subtree */
    while (pRepl->left != NULL)
    {
      pReplParent = pRepl;
      pRepl = pRepl->left;
    }
    if (pReplParent != p)
    {
      pReplParent->left = pRepl->right;
      pRepl->right = p->right;
    }
    pRepl->left = p->left;
  }
  free(p);
  printf("\nDeletion of node %d completed\n", key);
  if (pParent == NULL)
    return pRepl; /* original root was deleted */
  else
  {
    if (direction == LEFT)
      pParent->left = pRepl;
    else
      pParent->right = pRepl;
    return root;
  }
}
void delTree(BSTNodeT *root)
{
  if (root != NULL)
  {
    delTree(root->left);
    delTree(root->right);
    free(root);
  }
}
int getInt(const char *message)
{
    int val;
    printf(message);
    scanf("%d", &val);
    fflush(stdin);
    return val;
}
char prompt(const char *message)
{
    printf(message);
    char ch = getchar();
    fflush(stdin);
    return ch;
}
int main()
{
  BSTNodeT *p;
  int n, key;
  char ch;

  ch = prompt("Choose insert function:\n\t[R]ecursive\n\t[N]onrecursive: ");
  printf("Number of nodes to insert= ");
  scanf("%d", &n);
  root = NULL;
  for (int i = 0; i < n; i++)
  {
    key = getInt("\nKey= ");
    if (toupper(ch) == 'R')
      root = recInsert(root, key);
    else
      insert(key);
  }
  printf("\nPreorder listing\n");
  preorder(root, 0);
  while ('\n' != prompt("Press Enter to continue"));
```

```
  printf("\nInorder listing\n");
  inorder(root, 0);
  while ('\n' != prompt("Press Enter to continue"));
  printf("\nPostorder listing\n");
  postorder(root, 0);
  while ('\n' != prompt("Press Enter to continue"));
  ch = prompt("Continue with find (Y/N)? ");
  while (toupper(ch) == 'Y')
  {
    key = getInt("Key to find= ");
    p = find(root, key);
    if (p != NULL)
      printf("Node found\n");
    else
      printf("Node NOT found\n");
    ch = prompt("Continue with find (Y/N)? ");
  }
  ch = prompt("Continue with delete (Y/N)? ");
  while (toupper(ch) == 'Y')
  {
    key = getInt("Key of node to delete= ");
    root = delNode(root, key);
    inorder(root, 0);
    ch = prompt("Continue with delete (Y/N)? ");
  }
  ch = prompt("Delete the whole tree (Y/N)? ");
  if (toupper(ch) == 'Y')
  {
    delTree(root);
    root = NULL;
    printf("Tree completely removed\n");
  }
  while ('\n' != prompt("Press Enter to exit program"));
}
```

5.4 Laboratory Assignments

5.4.1 Mandatory Assignments

5.1. Study, understand, and test the supplied code.

5.2. Develop a modular program which demonstrates the BST operations in a tree which contains as keys the uppercase letters of the Roman alphabet. The operations are:

- insert, coded in input as `i<letter>` (e.g. `iD`) attempts to insert letter `D`;
- delete, coded in input as `d<letter>` (e.g. `dF`) attempts to delete node with key `F`;
- find, coded in input as `f<letter>` (e.g. `fG`) attempts to find node with key `G`;
- traversals (pre, in, post-order), coded in input as `t<letter>` (e.g. `tp` attempts to list nodes in preorder, `tP` attempts to list nodes in postorder, `ti`) attempts to list nodes in inorder;
- getting the minimum and maximum specified in input as `gm` (minimum), `gM` (maximum);

Use files, with names given on the command line for input and output.

5.3. A pharmacy keeps track of their supplies using the following fields: `name`, `price`, `amount`, `dateOfManufacturing`, `dateOfExpiration`. You should develop a program which uses BSTs as a supporting data structure in storing supply information for a pharmacy. The operations to implement are:

- insert, coded in input as `i<name>,<price>,<amount>,<dateOfManufacturing>, <dateOfExpiration>` (e.g. `iAspirin Bayer,7.65,6,20150702,20190702`) attempts to insert a product named "Aspirin Bayer", with a price of 7.65 monetary units, of which there are 6 boxes, manufactured in 2015, July, 2, and expiring in 2019, July, 2);
- find a medicine by name, coded as `u<name>`, and update fields not containing a single tilde`~`. E.g. `uAspirin Bayer,Aspirin another,4.5,~,~,20180405` will attempt to find the product with name `Aspirin Bayer`, and if found, update its name to `Aspirin another`, its price to 4.5, leave amount and manufacturing date unchanged and update expiration to 2018, April, 5;
- delete a product based on its name, if found. Operation is coded as `d<name>`. E.g `dAspirin another`;
- create another tree with expired products, encoded as `ce`.

Use files, with names given on the command line for input and output.

5.4.2 Optional Assignments

5.4. A binary search tree contains integer keys. develop a program which creates the BST from the keys provided as input. Then:

- find two nodes whose sum of key values is a given value, and print their values. The operation is specified in input as `s<amount>` (see below).
- find the path (if any) to traverse from given node to another. The operation is specified in input as `t<key1>,<key2>` (see below).

.

I/O description. Input: first line contains the list of keys to insert in the BST. Second line is the sum.

```
28␣41␣32␣17␣-3␣28␣107
s25
t17,107
```

Output

```
sum=25+-3
trav=17,28,41,107
```

5.5. Develop a program which reads words (strings of characters containing no whitespace[2] from an input file, and creates a search tree holding those words and their occurrence frequencies in its nodes. Print the words and their occurrence frequencies in ascending lexicographic order.

5.6. Implement the merge operation for two search trees. After reading the second list output should be automatically issued.

5.7. Write the functions find and insert for a search tree which holds character strings. Your routines should allow for wildcards – e.g. you can use '*' to specify any character string and '?' for specifying a single character. .
I/O description. Input may be read as triplets:

```
(<parent>␣<left>␣<right>)
```

A query is a pair of character strings, separated by whitespace, which does not begin with a '(', and, as specified above, may contain wildcards.

[2] blanks, tabs