4 Trees

4.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing arbitrary trees, and binary trees including balanced.
- To learn to compare the various simple tree models and choose the most suitable one for the given problem.
- To get hands-on experience with implementing trees, using dynamically allocated memory.
- To develop critical thinking concerning implementation decisions for trees .

The outcomes for this session are:

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

4.2 Brief Theory Reminder

The *binary* tree – a tree in which each node has at most two descendants – is very often encountered in applications. The two descendants are usually called the $left$ and $right$ children.

4.3 Operations on Binary Trees

Creating a Binary Tree

In order to create a binary tree, we can read the necessary information, presented in $preorder$ from the standard input. We have to distinguish between empty and nonempty (sub)trees, and thus we have to mark the empty trees with a special sign, such as '*'. For the tree of Figure 4.1, the description (on the standard input) is: `ABD*G***CE**F*H**`

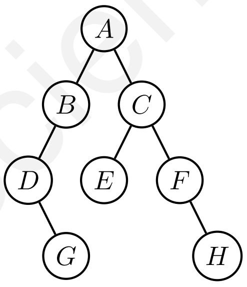

Figure 4.1: An example of a binary tree

The node structure is shown in Listing 4.1.

Listing 4.1: Example binary tree node structure

```
typedef struct node_type
{
  char id; /* node name */
  label ; /* appropriate type for label */
  struct node_type *left, *right;
} NodeT;
```

The construction of a binary tree may be achieved with a function similar to the one presented in Listing 4.2.

Listing 4.2: Creating a binary tree like the one in Figure 4.1

```
NodeT *createBinTree()
{
  NodeT *p;
  char c;
  /* read node id */
  scanf("%c", &c);
  if ( c == '*' )
    return NULL; /* empty tree; do nothing */
  else /* else included for clarity */
  { /* build node pointed to by p */
    p = ( NodeT *) malloc( sizeof( NodeT ));
    if ( p == NULL )
      fatalError( "Out of space in createBinTree" );
    /* fill in node */
    p->id = c;
    p->left = createBinTree();
    p->right = createBinTree();
  }
  return p;
}
```

The function *createBinTree* may be invoked as `root = createBinTree();`

Binary Tree Traversals

There are three kinds of traversals: preorder, inorder, and postorder. Listing 4.3 shows the C implementation for the operations of construction and traversal of a binary tree.

Listing 4.3: Construction and traversals of binary trees

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

typedef struct node_type
{
  char id; /* node name */
  /* ... other useful info */
  struct node_type *left, *right;
} NodeT;
void fatalError( const char *msg )
{
 printf( msg );
 printf( "\n" );
 exit ( 1 );
}
void preorder( NodeT *p, int level )
{
  if ( p != NULL )
  {
    for ( int i = 0; i <= level; i++ ) printf( " " ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    preorder( p->left, level + 1 );
    preorder( p->right, level + 1 );
  }
}
void inorder( NodeT *p, int level )
{
  if ( p != NULL )
  {
    inorder( p->left, level + 1 );
    for ( int i = 0; i <= level; i++ ) printf( " " ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    inorder( p->right, level + 1 );
  }
}
```

Listing 4.4: Construction and traversals of binary trees (continued)

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

  /* read node id */
  if ( branch == 0 )
     printf( "Root identifier [0 to end] =" );
  else
  if ( branch == 1 )
     printf( "Left child of %d [0 to end] =",
              parent->id );
  else
     printf( "Right child of %d [0 to end] =",
              parent->id );
  scanf("%d", &id);
  if ( id == 0 )
    return NULL;
  else
  { /* build node pointed to by p */
    p = ( NodeT *)malloc(sizeof( NodeT ));
    if ( p == NULL )
      fatalError( "Out of space in createBinTree" );
    /* fill in node */
    p->id = id;
    p->left = createBinTree( 1, p );
    p->right = createBinTree( 2, p );
  }
  return p;
}
int main()
{
  NodeT *root = createBinTree( 0, NULL );
  while ('\n' != getc(stdin));
  printf( "\nPreorder listing\n" );
  preorder( root, 0 );
  printf( "Press Enter to continue." );
  while ('\n' != getc(stdin));
  printf( "\nInorder listing\n" );
  inorder( root, 0 );
  printf( "Press Enter to continue." );
  while ('\n' != getc(stdin));
  printf( "\nPostorder listing\n" );
  postorder( root, 0 );
  printf( "Press Enter to continue." );
  while ('\n' != getc(stdin));
  return 0;
}
```

4.4 Balanced Binary Trees

A *balanced* binary tree is a binary tree where the number of nodes in any of the left subtrees is at most one more than the number of nodes in the corresponding right subtree.
The following algorithm can be used to build such a tree with n nodes:

1. Choose a node to be the root of the tree.
2. Set the number of nodes in the left subtree to $n_{left} = \frac{n}{2}$.
3. Set the number of nodes in the right subtree to $n_{right} = n - n_{left} - 1$.
4. Repeat the steps recursively, taking the number of nodes to be n_{left} till there are no more nodes.

5. Repeat the steps recursively, taking the number of nodes to be n_{right} till there are no more nodes.

Listing 4.5: Construction and traversals of completely balanced binary trees

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

typedef struct node_type
{
  char id; /* node name */
  ... /* other useful info */
  struct node_type *left, *right;
} NodeT;
void fatalError( const char *msg )
{
 printf( msg );
 printf( "\n" );
 exit ( 1 );
}
void preorder( NodeT *p, int level )
{
  if ( p != NULL )
  {
    for (int i = 0; i <= level; i++ )
      printf( " " ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    preorder( p->left, level + 1 );
    preorder( p->right, level + 1 );
  }
}
void inorder( NodeT *p, int level )
{
  if ( p != NULL )
  {
    inorder( p->left, level + 1 );
    for (int i = 0; i <= level; i++ )
      printf( " " ); /* for nice listing */
    printf( "%2.2d\n", p->id );
    inorder( p->right, level + 1 );
  }
}
void postorder( NodeT *p, int level )
{
  if ( p != NULL )
  {
    postorder( p->left, level + 1 );
    postorder( p->right, level + 1 );
    for ( int i = 0; i <= level; i++ )
      printf( " " ); /* for nice listing */
    printf( "%2.2d\n", p->id );
  }
}
NodeT *createBalBinTree( int nbOfNodes )
{
    NodeT *p;
    if ( nbOfNodes == 0 ) return NULL;

    int nLeft = nbOfNodes / 2;
    int nRight = nbOfNodes - nLeft - 1;
    p = ( NodeT * ) malloc( sizeof( NodeT ));
    if (p)
    {
        printf( "Node identifier in preorder=" );
        scanf( "%d", &p->id );
        p->left = createBalBinTree( nLeft );
        p->right = createBalBinTree( nRight );
    }
    else abort(); // no memory left
    return p;
}
```

Listing 4.6: Construction and traversals of balanced binary trees (continued)

```
int main()
{
  int nbOfNodes = 0;

  printf("\nNumber of nodes in the tree=");
  scanf("%d", &nbOfNodes );
  NodeT *root = creBalBinTree( nbOfNodes );;

  while ('\n' != getc(stdin));
  printf( "\nPreorder listing\n" );
  preorder( root, 0 );
  printf( "Press Enter to continue." );
  while ('\n' != getc(stdin));
  printf( "\nInorder listing\n" );
  inorder( root, 0 );
  printf( "Press Enter to continue." );
  while ('\n' != getc(stdin));
  printf( "\nPostorder listing\n" );
  postorder( root, 0 );
  printf( "Press Enter to continue." );
  while ('\n' != getc(stdin));
  return 0;
}
```

4.5 Arbitrary Trees

An *arbitrary* tree is a tree where the interior nodes have more than two children.
A node of such a tree may be structured as in 4.7;

Listing 4.7: Example of arbitrary tree node record

```
/* maximum number of children */
#define MAX_CHILD <appropriate value>
typedef struct node_type
{
  char id; /* node name */
  ... /* other useful info */
  struct node_type *children[MAX_CHILD];
} NodeT;
```

To build such a tree:

1. Read for each node, in *postorder*, the fields: `id`, other useful info[1], and the number of children and push this info onto a stack.
2. When parent node is read, pop children nodes of the stack, fill children pointers in parent node, and then push a pointer to parent onto stack. Finally the only pointer on the stack will be a pointer to the root node, and the process stops.

Tree traversal is achieved in level order, according to the following:

- Use a queue to keep pointers to nodes to be processed. The queue is initially empty.
- Enqueue a root node pointer
- Dequeue a node, process node information, and enqueue pointers to the children of the currently processed node.
- Repeat previous step till the queue becomes empty.

4.6 Laboratory Assignments

Study the provided code and use that code and code derived from it to solve the mandatory problems.

For all proposed problems, input and output data should reside in files whose names are given as command line arguments.

Here are some issues to think about:

- What are the pros and cons for using static vs. dynamically allocated memory for storing a tree?

[1] if defined

- Why should we try to achieve a more general (abstract) implementation of a tree instead of "solving" the problem at hand?
- What are the benefits of using a modular approach instead of a monolithic one in our implementation?
- How can we test our implementation?
- How can we generate test data to use in checking that our program works properly?

4.6.1 Mandatory Assignments

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

4.2. Modify the supplied code and add functions to:

- Interchange the left and right subtrees of a given node. Output the traversals of the new tree in preorder, inorder and postorder.
- Determine the height of a binary tree, Output: height as a number.
- Determine the number of leaves in a binary tree. Output: number of leaves.

4.6.2 Optional Assignments

4.3. Read a mathematical expression, in postfix form, as a character string, from the standard input device. Build the tree for that expression. Every node should contain either an operator or an operand. Here, `a, c, d, e` stand for integer numbers. Thus, is `a=-2, c=3, d=7` and `e=-11` the actual expression would be `-2 3 + 7 - -11 *` .

I/O description. Input:

```
e:a␣c␣+␣d␣-␣e␣*
p
```

where `e:`=expression definition follows, `p:`=request expression print.

- Print the expression in postfix, prefix and infix form.
- Evaluate the expression. Command: `e=`, Value should be output.

4.4. A similar problem, but expression if composed of boolean values with 0=false, and 1=true, and the allowed operators are: AND, OR and NOT. Same commands.

4.5. Construct a family tree organized as follows: the root node holds a name as a key; the key of the left child node holds father's name, the right child node holds mother's name. Each family is restricted to have a single child. Then, if pairs of names are read from the standard input, output the relations between the two persons. .
I/O description. Input

```
(<childName>␣(<motherName>␣<fatherName>))
...
?<name1>,␣<name2>
```

All names (i.e. `<childName>, <motherName>, <fatherName>, <name1>, <name2>` are alphabetic strings with no spaces. Parenthesis, i.e. "`(`", "`)`" are delimiters. "`?`" means that line contains a query about persons. Output is relation degree (i.e, child, parent, cousin, etc.). What if unrelated trees occur? How would you store the forest? Figure 4.2 illustrates a family tree. Note that the ancestors are at the leaf level.

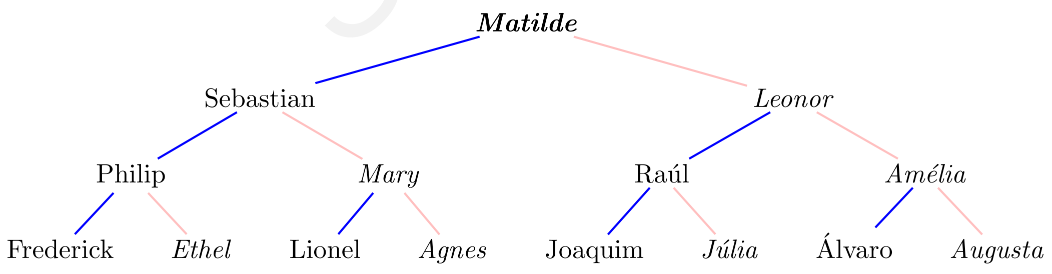

Figure 4.2: Example family tree

.

I/O description. For the following problems, tree input data may organized in lines as:

```
parent left right
```

4.6. Transform a binary tree into a doubly-linked list .
I/O description. Output the traversal of the list.
4.7. Find out if two binary tree are *equivalent* (i.e. equivalent as structure and the corresponding nodes hold the same data). .
I/O description. Output $\in \{yes, no\}$.
4.8. Construct and traverse an arbitrary tree built as described in §4.5. Output is pre-, post- and in- order traversal.