

School of Computer Science and Software Engineering
Clayton Campus, Monash University
CSE1303 Part A
Summer Semester, 2002

Practical Session 5: Binary Search Trees



Aims of this Practical Session

1. To familiarise yourself with Binary Search Trees and how to implement them.
2. To familiarise yourself with insertion sort and how to implement this algorithm.

Project: To write a program to find anagrams.

Background:

An **anagram** of a word is a rearrangement of the letters in the word, in order to form a different word. For example **least**, **stale**, and **slate** are all anagrams of each other – they all use the same letters, just in a different order.

In this practical session you will use a Binary Search Tree to implement a database for storing character strings. Then you will use this database in a program to find anagrams.



Note: All example input files mention below can be found can be found at:

<http://www.csse.monash.edu.au/courseware/cse1303s/parta/prac05/>

Preparation (3 marks if completed before class)

Before this prac:

1. Find the anagrams of:
 - (a) tear
 - (b) dog
2. Modify the code for a Binary Search Tree given in lectures, so that instead of holding numbers of type **float** it holds an **address** of a character string.
 - ⊕ **You must use the function `strdup` to make a copy of each character string in function `makeNode`. If your C libraries do not contain a copy of `strdup`, use the one given in the solutions for Tute 3.**
3. Write a C program which uses a Binary Search Tree to store character strings, and has a menu with the following options:
 - **Read String:** which asks the user for a character string, and stores the character string in the Binary Search Tree.
 - **List:** which prints out all the character strings stored in the Binary Search Tree in alphabetical order.
 - **Quit:** which allows the user to quit the menu and exit the program.
 - ⊕ **You may assume all character strings have at most 30 characters.**
 - ⊕ **Deallocate any memory allocated when it is no longer required.**

Question 1: (2 marks)

Modify your C program you wrote for the preparation to have the following functions and options:

- **Read File:** which asks the user for a filename, opens the file, and stores all the character strings in the file in the Binary Search Tree.
- **Search:** which asks the user for a character string, and then prints a message indicating whether or not the string is contained in the Binary Search Tree.
- **List:** which prints out all the character strings stored in the Binary Search Tree in alphabetical order.
- **Quit:** which allows the user to quit the menu and exit the program.

Question 2: (2 marks)

Write the following two C functions and a C program which demonstrates that the two functions work.

- **char* doubleString(char* str)** which creates a new character string which contains a copy of **str**, followed by a space, followed by another copy of **str** concatenated together, and returns the address of the new string. For example, when given the string “**spot**” it return the address to the character string , “**spot spot**”.
- **void sortString(char* str, int size)** which sorts the letters of the string, of length **size**, into alphabetical order using insertion sort. For example, when given the string “**spot**” and **4** as input, on return the string would be “**opst**”

Question 3: (3 marks)

Change the program you wrote in Question 1, so that the program has a menu with the following options:

- **Read** which asks the user for a filename and opens the file. Then for each word in the file stores into the Binary Search Tree a character string, consisting of:
 - the letters of the word sorted in alphabetical order,
 - followed by a space,
 - followed by a copy of the word.

For example instead of storing the words **spot**, **pots**, **act**, and **cat**, it inserts the strings, “**opst spot**”, “**opst pots**”, “**act act**”, and “**act cat**” into the Binary Search Tree. (*use the function you wrote for Question 2b to do the sorting*)

- **Search**: which asks the user for a word, and then prints the second half of any string (the part after the space) in the Binary Search Tree whose first half is the word sorted alphabetically.

For example if “**opst spot**”, “**opst pots**”, “**act act**”, and “**act cat**” were the only strings contained in the Binary Search Tree, and the user gave the word , “**spot**”, the program should print “**spot**” and “**pots**”.

❖ **Note this option is finding the anagrams of the given word.**

- **Quit**: which allows the user to quit the menu and exit the program.

⊕ **Test your program on the file words.txt, which can found at <http://www.csse.monash.edu.au/courseware/cse1303s/parta/prac/prac05/words.txt>, and find the anagrams of subessential.**

Advanced Questions: (2 marks)

Consider the problem of traversing a Binary Search Tree and performing some operation on each node. One way of doing this is as follows.

(Refer to section 7.11 (pp201-4) of Deitel & Deitel 2nd Edition for function pointers)

```
void
traverseInorder(TreeNode* nodePtr, void (*Op)(TreeNode*))
{
    if (nodePtr != NULL)
    {
        traverseInorder(nodePtr->leftPtr, Op);
        (*Op)(nodePtr);
        traverseInorder(nodePtr->rightPtr, Op);
    }
}
```

The parameter **Op** in this function is a pointer to a function, and the function that **Op** points to performs the operation that is required. For example, imagine that the nodes in the Binary Search Tree have a key which is of type **char***, and that we have the following function.

```
void
printKey(TreeNode* nodePtr)
{
    printf("%s\n", nodePtr->key);
}
```

Then the function call

```
traverseInorder(rootPtr, printKey);
```

will print all the keys in all the nodes in the subtree whose root is pointed to by **rootPtr**.

Now consider the problem of searching for a node in a Binary Search Tree whose **key** has a property in common to a given string. For example, the first half of the **key** might be identical to the string sorted alphabetically.

One way of solving this problem is write a general search function, which has the parameters **nodePtr**, **str**, and **cmp**, where:

- **nodePtr** contains the address of the current node,
- **str** is the address of the given character string, and
- **cmp** is a pointer to a function that compares two character strings and returns a negative value if the first argument is less than the second, zero if they are equal, and a positive value if its greater.

Questions:

- Write a general search function

TreeNode* search(TreeNode* nodePtr, char* str, int (*cmp)(char*, char*))

- Rewrite the program you wrote for Question 3, using the general search function, and different comparison functions.