Exam 2

MIDTERM EXAM 2

Note: Midterm 2 contains 5 questions. Questions written on paper should be photographed and uploaded as JPEG or PDF. Question 5 should be submitted as C++ source file in a separate folder.

Question 1:

Alice sends messages to Bob using only eight voiced consonants from this alphabet: {B, D, G, J, N, R, V, Z}. Each consontant is encoded as a sequence of bits (0s and 1s) using the binary tree shown below:

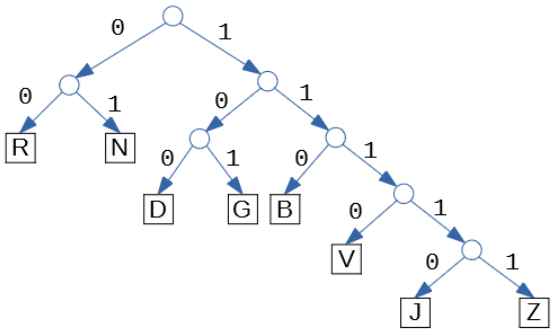


Fig. 1: A Binary Tree used by Alice to encode 8 consonant letters.

For example, VZJ is encoded as 1110.11111.11110 (V becomes 1110, Z becomes 11111 and J becomes 11110). Each code is obtained by following the edges from the root to the respective leaf in this tree.

(A) Show how Alice can encode the following 8-letter word: BRBDNGNG. How many bits does it use? (Please separate the codes of individual letters with dots for better readability.)

(B) In Alices language the probabilities of letters are the following:

R	N	D	G	B	V	J	Z
32%	28%	14%	11%	9%	4%	1%	1%

Find the expected value of the total number of bits (0s and 1s) that she uses in order to send a random 10-letter word to Bob. (The letters in a random word are chosen independently according to the probabilities given above).

Answer:

(A) BRBDNGNG becomes the following sequence of bits:

110.00.110.100.01.101.01.101

The total length of this sequence is 21 bits (not counting the separating dots). It is indeed sufficient to send just this sequence: 110001101000110101101 for Bob to be able to decode it.

Note: Even without the separating dots this sequence can be deciphered: Every time we start at the root of the tree and see where it finishes. For example, the sequence 110 ends with a leaf denoting letter B (it cannot be continued as the code 110 is not a prefix for any other letter). Such codes are named *prefix codes* and the method to build an efficient prefix tree is called *Huffman coding*.

(B) Each of the letters a has a fixed-length code; let us denote it by $\ell(a)$. Let X be a random variable that shows the number of bits used to encode a *single* letter from that alphabet of 8 letters. We can find $E(X)$ by adding code lengths multiplied by letter probabilities:

$$\begin{aligned} E(X) &= P(\mathtt{R}) \cdot \ell(\mathtt{R}) + P(\mathtt{N}) \cdot \ell(\mathtt{N}) + P(\mathtt{D}) \cdot \ell(\mathtt{D}) + P(\mathtt{G}) \cdot \ell(\mathtt{G}) + \\ &+ P(\mathtt{B}) \cdot \ell(\mathtt{B}) + P(\mathtt{V}) \cdot \ell(\mathtt{V}) + P(\mathtt{J}) \cdot \ell(\mathtt{J}) + P(\mathtt{Z}) \cdot \ell(\mathtt{Z}) = \\ &= 0.32 \cdot 2 + 0.28 \cdot 2 + 0.14 \cdot 3 + 0.11 \cdot 3 + 0.09 \cdot 3 + 0.04 \cdot 4 + 0.01 \cdot 5 + 0.01 \cdot 5 = \\ &= 2.48 \end{aligned}$$

The expected value for encoding 10 letters is ten times larger: 24.8.

Question 2:

Some binary tree T has exactly 32 internal nodes.

(A) Can tree T be a full binary tree? (In a full binary tree every node has either two children or no children at all.) Can tree T be a perfect binary tree? (In a perfect binary tree all leaves have the same depth.)

(B) What is the largest and the smallest value for n – the total number of nodes in the tree T? Explain your estimates.

(C) What is the largest and the smallest value for h – the height of T? Explain your estimates.

Answer:

(A) The tree can be a full binary tree. Consider the following construction: first create the simplest full binary tree only containing one node (it is its root, but it is also a leaf). Every time some leaf gets two children, it becomes an internal node and there are two new leaves – in total the tree gains one internal node and one leaf node. So, the number of leaves is always larger than the number of internal nodes (by exactly one). So a tree with 32 internal nodes would have exactly 33 leaves.

No tree with 32 internal nodes can be a perfect binary tree. In perfect trees the number of internal nodes can be 0 or 1, or 3, or 7, and so on. In general, a perfect tree of height H would have exactly $2^H - 1$ internal nodes (and 2^H leaves). But 32 cannot be represented as $2^H - 1$ for any integer H.

(B) If a tree has i internal nodes, then it can have up to $i + 1$ leaves (it has exactly $i + 1$ leaves, if it is a full tree – see explanation in the above item). On the other hand, any tree with i internal nodes should have at least one leaf – the node where parent-child relationships end. A tree can have exactly one leaf (if any internal node has exactly one child – so all the nodes make a long chain with i internal nodes and one leaf).

In case if $i = 32$ we get the maximum number of nodes $n = i + (i + 1) = 32 + 33 = 65$. And the minimum number of nodes is $i + 1 = 33$.

(C) Consider a perfect tree with all leaves at the depth 5. It would have $2^5 = 32$ leaves, but just 31 internal nodes. To create one more internal node we must have at least two leaves at the depth 6, so the smallest value of h is $h = 6$.

On the other hand, the very skinny tree – just a chain of vertices with one leaf at the end – would have height $h = 32$. (One cannot get a taller tree, since there must be at least one vertex at every depth; one cannot skip levels.)

Question 3:

Minimum Priority Queue has this ADT (Abstract Data Type):

PQ.getEmpty()	$\Theta(1)$	// initialize `PQ` to an empty priority queue
void PQ.insert(E item)	$\Theta(\log_2 n)$	// insert `item` into the priority queue `PQ`.
E PQ.min()	$\Theta(1)$	// return an item with minimum key value, do not modify `PQ`.
void PQ.removeMin()	$\Theta(\log_2 n)$	// remove an item with minimum key from `PQ`
int PQ.size()	$\Theta(1)$	// return the number of items in the priority queue `PQ`

Every function in this ADT has its time complexity written in the 2nd column – it corresponds to the heap implementation.

Consider the following pseudocode. Denote the number of items in the original list `L` by n (assume that $n \geq 10$ and all items in this list have different keys).

```
ProcessList(L):
    PQ.getEmpty()
    foreach item in L:
        PQ.insert(item)
    while PQ.size() > 5:
        PQ.removeMin()
    return PQ.min()
```

(A) Describe in English what does the function ProcessList(L) return.

(B) Express the time complexity of this algorithm in Big-Θ notation: Find a function $g(n)$ such that the time complexity of ProcessList(L) is $\Theta(g(n))$.

Answer:

(A) The algorithm keeps removing items until $PQ.size() == 5$. This means that only the five largest items remain in the priority queue. After that we return the minimum of the remaining elements.

For this reason the function ProcessList(L) will return the 5th largest element of the list L.

(B) For large values of n there will be $n - 5$ operations $PQ.removeMin()$. Each operation takes $O(\log_2 n)$ time. Therefore the total time of this algorithm is $O(n \log_2 n)$. Big-O notation is the upper estimate.

Let us show that also the lower estimate (Big-Omega notation) is $\Omega(n \log_2 n)$. One could argue that as the queue becomes shorter the $removeMin()$ will gradually become faster. But this does not change the lower estimate of time complexity, as there will be at least one half of the $removeMin()$ calls – namely, $n/2$ calls; and every one will operate on a heap of size at least $n/2$, and its logarithm is $\log_2(n/2) = \log_2 n - 1$. Product of $n/2$ and $(\log_2 n - 1)$ is $\Omega(n \log_2 n)$.

Since both estimates are the same, the complexity of algorithm is $\Theta(n \log_2 n)$.

Question 4:

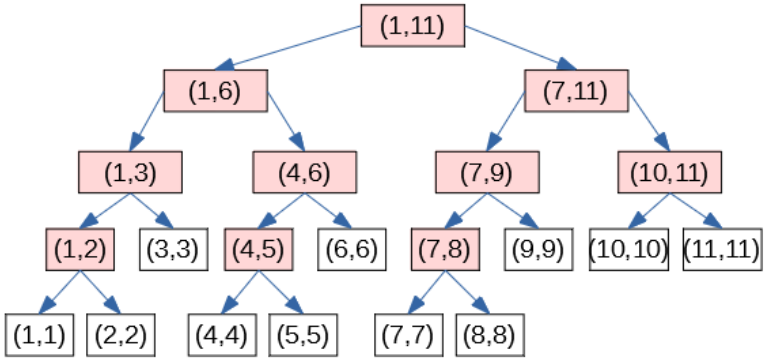
We have a 1-based array with 11 elements: $A[1], \ldots, A[11]$. We want to sort it efficiently. Consider the following Merge sort pseudocode:

MERGESORT(A, p, r):
1 **if** $p < r$
2 $\quad q = \lfloor (p + r)/2 \rfloor$
3 $\quad$ MERGESORT(A, p, q)
4 $\quad$ MERGESORT$(A, q + 1, r)$
5 $\quad$ MERGE(A, p, q, r)

Assume that initially you call this function as MERGESORT(A,1,11), where $p = 1$ and $r = 11$ are the left and the right endpoint of the array being sorted (it includes both ends).

What is the total number of calls to MERGESORT for this array (this includes the initial call as well as the recursive calls on lines 3 and 4 of this pseudocode).

Answer:



The recursive calls of MERGESORT are shown in the figure – just the parameters p, r for each call. For example, MERGESORT$(A, 1, 11)$ computes $q = \lfloor (1 + 11)/2 \rfloor = 6$, and causes two more calls to MERGESORT$(A, 1, 6)$ and MERGESORT$(A, 7, 11)$ respectively. On the other hand, if $p = r$, then the recursive calls do not happen (one-element list is already sorted). So there are exactly 11 external nodes (leaves) in the recursion tree.

Since the tree of calls is full, it also has 10 internal nodes (shown pink in the picture). The total number of these nodes is $10 + 11 = 21$.

Question 5:

Complete the C++ program that converts a tree into a string using function `toString()`. A tree can be built from two types of objects: objects of class `Leaf` and objects of class `Internal`. They are both inherited from a common parent class `Node`. All nodes have attribute `label`. Moreover, `Internal` nodes have attribute `children` of type `list<Node*>` – it is a list of pointers to the child nodes (either leaves or other internal nodes).

```
class Node {
public:
    string label;
    virtual string toString() = 0;
};

class Leaf : public Node {
public:
    Leaf(string arg) { label = arg; }
    virtual string toString()
    {
      // TODO: Insert your code here
```

(continues on next page)

(continued from previous page)

```
    }
};

class Internal : public Node {
public:
    list<Node *> children;
    Internal(string arg, list<Node *> ch)
    {
        label = arg;
        children = ch;
    }
    virtual string toString()
    {
      // TODO: Insert your code here
    }
};

int main() {
    Node *tree1Root = new Leaf("AAA");
    cout << tree1Root->toString() << endl << endl;

    Node *tree2Root = new Internal("AA",
        {new Leaf("BB"), new Leaf("CC")});
    cout << tree2Root->toString() << endl << endl;

    Node *tree3Root = new Internal("A",
        {new Internal("B", {
            new Internal("C", {new Leaf("D"), new Leaf("E")}),
            new Internal("F", {new Leaf("G"),
                new Internal("H", {new Leaf("I"), new Leaf("J")})}),
        }),
        new Internal("K", {new Internal("L", {new Leaf("M")})})});
    cout << tree3Root->toString() << endl
         << endl;
}
```

A tree that is a leaf is converted to a string: the value of `label`. A tree that is an internal node is converted to a string as the parents label followed by all the subtrees under that parent – they are all separated by single spaces and enclosed in parentheses.

Here is the expected output from the program:

```
AAA

(AA BB CC)

(A (B (C D E) (F G (H I J))) (K (L M)))
```

In this task you can complete the functions `toString()` and possibly make other changes. You should not modify the method `main()`; your code should preserve the inheritance relations between `Node`, `Leaf` and `Internal`. Solutions that use hard-coded output that does not depend on the values and structures defined in `main()` will not be considered valid.

Answer:

A possible solution to this task is shown in the code below:

```
#include <iostream>
#include <list>
#include <sstream>
#include <string>

using namespace std;
class Node
{
public:
    string label;
    virtual string toString() = 0;
};

class Leaf : public Node
{
public:
    Leaf(string arg) { label = arg; }
    virtual string toString()
    {
        stringstream ss;
        ss << label;
        return ss.str();
    }
};

class Internal : public Node
{
public:
    // string label;
    list<Node *> children;
    Internal(string arg, list<Node *> ch)
    {
        label = arg;
        children = ch;
    }
    virtual string toString()
    {
        stringstream ss;
        ss << "(";
        ss << label;
        for (Node *child : children)
        {
            ss << " " << child->toString();
        }
        ss << ")";
        return ss.str();
    }
};

int main()
{
    Node *tree1Root = new Leaf("AAA");
    cout << tree1Root->toString() << endl << endl;

    Node *tree2Root = new Internal("AA", {new Leaf("BB"), new Leaf("CC")});
    cout << tree2Root->toString() << endl << endl;
```

(continues on next page)

(continued from previous page)

```
    Node *tree3Root = new Internal("A",
        {new Internal("B", {
            new Internal("C", {new Leaf("D"), new Leaf("E")}),
            new Internal("F", {new Leaf("G"),
                new Internal("H", {new Leaf("I"), new Leaf("J")})}),
        }),
        new Internal("K", {new Internal("L", {new Leaf("M")})})});

    cout << tree3Root->toString() << endl
         << endl;

    return 0;
}
```

Make sure that you declare `toString()` function in the superclass `Node` as virtual. Also – it may be convenient to use `stringstream` class to accumulate output (and only at the very end convert it to a C++ `string` object). Another alternative would be – concatenate many strings in the function `Internal::toString()`.