Data Structures And Algorithms

Office : LWSN 3151-1
Office Hours : Tuesdays 10-12pm
Topics

- Recursion
  - List reversal
  - Prefix → Postfix
- Heap
  - Insertion()
  - DeleteMin()
- Huffman codes
  - Implementation
Recursion

- **Base case:**
  - Think of the simplest input to the problem that you know the answer to.

- **Recursive step:**
  - Think of how you would solve the problem in terms of a smaller input. Do part of the work now, then make a recursive call to handle the rest.
Recursion examples

input: integer n such that n >= 0
output: [n × (n-1) × (n-2) × … × 1]

factorial (int n)
  1. if n is 0, return 1
  2. otherwise, return [ n × factorial(n-1) ]
end factorial
Recursion examples

- Reverse a List recursively
Recursion examples

- Recursively convert prefix expression to postfix expression.
Prefix → Postfix

- **Prefix expression**
  
  \[
  \text{PREFIX} \rightarrow \text{IDENTIFIER} \mid \text{OPERATOR} \text{PREFIX} \text{PREFIX}
  \]
  
  \[
  \text{OPERATOR} \rightarrow + \mid - \mid * \mid /
  \]
  
  \[
  \text{IDENTIFIER} \rightarrow a \mid b \mid c \mid \ldots \mid z
  \]

- **Postix expression**
  
  \[
  \text{POSTFIX} \rightarrow \text{IDENTIFIER} \mid \text{POSTFIX} \text{POSTFIX} \text{OPERATOR}
  \]
  
  \[
  \text{OPERATOR} \rightarrow + \mid - \mid * \mid /
  \]
  
  \[
  \text{IDENTIFIER} \rightarrow a \mid b \mid c \mid \ldots \mid z
  \]
int prefixToPostfix(const string& prefix, string& postfix)
{
    // BASE CASE : PREFIX → IDENTIFIER
    char firstChar = prefix[0];
    if (isOperand(firstChar))
    {
        // must be: <operand>
        postfix += firstChar;
        return 1;
    }

    // Rec step: PREFIX → OPERATOR PREFIX PREFIX
    int nextPrefix1 = pre2post(prefix.substr(1), postfix);
    int nextPrefix2 = pre2post(prefix.substr(nextPrefix1 + 1), postfix);
    postfix += firstChar; // the operator
    return nextPrefix1 + nextPrefix2 + 1;
}
Topics

- Recursion
  - List reversal
  - Prefix → Postfix
- Heap
  - Insertion()
  - DeleteMin()
- Huffman codes
  - Implementation
Heaps

- Almost complete binary trees
- Stores key-element pairs
- MinHeap: key(parent) <= key(child)
Heaps - operations

- Insert()
- DeleteMin()
Heaps - Insert

- Inserting '6'
Heaps - Insert

- Inserting '6' in next available position
Heaps - Insert

![Diagram of heaps with nodes and values]

The diagram shows the process of inserting a new element into a heap, maintaining the heap property.

- The tree structure is annotated with node values and arrows indicating the path taken during the insert operation.
- The heap property is preserved after the insertion.

This visual representation helps in understanding how heaps are structured and how insertions are carried out to maintain the heap property.
Heaps - Insert
Heaps - Insert
Heaps - Insert
Heaps - DeleteMin

- Remove the element from heap with minimum key
Heaps - DeleteMin
Heaps - DeleteMin
Heaps - DeleteMin
Heaps - DeleteMin
Topics

• Recursion
  ▪ List reversal
  ▪ Prefix → Postfix

• Heap
  ▪ Insertion()
  ▪ DeleteMin()

• Huffman codes
  ▪ Implementation
Huffman codes

Informal description

**Given**
A set of symbols and their weights (or frequencies).

**Find**
A *prefix-free binary code* (a set of codewords) with minimum codeword length (equivalently, a tree with minimum weighted path length from the root).
Huffman codes

- Insert all the characters in the priority queue.
- While (size(queue) > 1):
  - Remove the two nodes of lowest frequency
  - Create a new internal node with these two nodes as children
  - Freq(newNode) = sum of frequencies of its children.
  - Add this new node to the queue.

Running time is O(n log n)
Huffman codes
Questions