Proyecto: Algoritmo Radix Sort en LISP

Objetivo: En este proyecto, el objetivo es implementar el algoritmo de ordenamiento Radix Sorten Lisp. Radix Sortes un algoritmo eficiente para ordenar enteros que explora los dígitos de los números para realizar la clasificación. Este proyecto les permitirá aplicar conceptos de estructuras de datos y algoritmos de ordenamiento en un contexto práctico, así como familiarizarse con Lisp.

Implementación:

(defvar cola '())

(defvar lista-colas (list (list) (list) (list) (list) (list) (list) (list) (list) (list) (list)))

(defvar lista-colas-neg (list (list) (list) (list) (list) (list) (list) (list) (list) (list) (list)))

(defun RADIX-Sort (cola multiplicador)

    (if (< multiplicador 100000)

        (progn

            (mapcar #'(lambda (x) (llenar-lista-colar x multiplicador)) cola)

            (mapcar #'llenar-colas-neg lista-colas-neg)

            (mapcar #'llenar-colas lista-colas)

            (setq lista-colas (list (list) (list) (list) (list) (list) (list) (list) (list) (list) (list)))

            (setq lista-colas-neg (list (list) (list) (list) (list) (list) (list) (list) (list) (list) (list)))

            (radix-sort cola (\* multiplicador 10))

        )

        (print (reverse cola))

    )

)

(defun llenar-lista-colar (numero multiplicador)

    (if (and (> (mod (truncate (/ (car cola) multiplicador)) 10) 0) (< (car cola) 0))

        (push (car cola)

            (nth (funcall (lambda (x) (+ (mod (truncate (/ x multiplicador)) 10) 0)) (car cola)) lista-colas-neg)

        )

        (push (car cola)

            (nth (funcall (lambda (x) (+ (mod (truncate (/ x multiplicador)) 10) 0)) (car cola)) lista-colas)

        )

    )

    (pop cola)

)

(defun llenar-colas (lista)

    (if (null lista)

        (pop lista-colas)

        (progn

            (push (car lista) cola)

            (llenar-colas (cdr lista))

        )

    )

)

(defun llenar-colas-neg (lista)

    (if (null lista)

        (pop lista-colas-neg)

        (progn

            (push (car lista) cola)

            (llenar-colas-neg (cdr lista))

        )

    )

)

(defun llamar-radix-sort (list)

    (radix-sort list 1)

)

(llamar-radix-sort '(1 -1 -10 9 1879 -8923 -23 0 -18 7823))

Informe técnico:

Algoritmo Radix sort:

El algoritmo Radix Sort es un método utilizado ampliamente en los sistemas de cómputo para poder ordenar datos ya sea de menor a mayor o viceversa, es bastante eficiente en casos específicos sobre todo al compararlo con otros métodos de ordenamiento más tradicionales.

La principal desventaja de este algoritmo es que para que su eficiencia sea la correcta, se debe conocer previamente los dígitos que tiene el número más grande o pequeño que será ordenado, ya que, la cantidad de ciclos dependerá de la cantidad de los dígitos que se tengan.

La manera principal de como funciona este algoritmo es de la siguiente manera:

1. Toma la primera unidad de todos los números y las mete en diferentes listas de acuerdo con su valor.
2. Une todas esas listas y las regresa a la lista original.
3. Repite el proceso para las decenas.

De esta manera el algoritmo logra ordenar todos los números, incluso si las unidades o los décimos, o cualquier otro se repite. Gracias a que no compara los valores, es bastante más eficiente que otros métodos, sin embargo, puede ser poco eficiente si no se conoce previamente los dígitos que tendrán los números ingresados, además, puede ser un tanto complicado implementarlo para tomar en cuenta a los números del conjunto de los negativos.

Para poder implementar este algoritmo en common Lisp, creamos 3 listas que las trataremos como colas, una es la principal, y tendremos otras 2 de 10 listas cada una, una para los positivos y otra para los negativos, luego se manda a llamar a un método que se encarga de implementar el algoritmo, el cuál funciona de la siguiente manera.

Primero recorre la lista aplicando la función llamada “Llenar-lista-colas” por cada uno de los elementos, este método se encarga de clasificar los números de acuerdo con su dígito, internamente este método clasifica los negativos y los positivos con la estructura condicional: (if (and (> (mod (truncate (/ (car cola) multiplicador)) 10) 0) (< (car cola) 0))))

Si solo hacemos la condición con la cabeza de la cola nos ocasionará problemas, dado que este será negativo a pesar de que el dígito no lo sea, por lo que la primera condición se encarga de asegurar que su modulo no sea 0, lo que significaría que aun existe un dígito entero, y la segunda parte de que sea negativo, de esa forma se le puede dar un manejo adecuado para los números enteros negativos.

Para los números positivos se le da otro tipo de manejo del que hablaremos en un momento, sin embargo, cabe destacar que si el modulo de un número negativo es 0, se deberá tratar en ese momento como un positivo 0, para que se acomoden de forma correcta los números negativos.

Iniciaremos hablando de como se tratan los números positivos, para ello se utiliza la siguiente línea de código:

(push (car cola)

            (nth (funcall (lambda (x) (+ (mod (truncate (/ x multiplicador)) 10) 0)) (car cola)) lista-colas)

        )

En esta línea se mete la cabeza de la cola (que es el número a analizar y ordenar) a la posición correspondiente de las 10 listas creadas según su unidad, es decir, si tenemos el número 127, se meterá en la lista número 8 (el 0 se cuenta como 1), gracias a esto se pueden ordenar los números basados en una especie de prioridad. Cabe destacar que el número una vez entra, no entra como unidad, más bien, entra en su forma original.

Se utiliza la función truncate para ignorar los decimales y que no causen inconvenientes en el código.

Al final, se sacan los elementos de la cola original mediante el método “pop” y así se continúa clasificando y rellenando las listas de listas según el valor de la unidad.

Si el valor es negativo, se realiza prácticamente el mismo proceso, sin embargo, se llena la lista de listas llamada “lista-colas-neg” la cual permite tratarlos como negativos, en donde hasta el momento no hay cambios.

Para continuar, se vacían las listas de listas en la lista original de forma ordenada, para ello se comienza con las listas negativas, ya que son las que van primero por ser menores, sus métodos lo único que hacen es copiar las listas en la lista original de forma recursiva para vacías todas y cada una de las listas de manera ordenada. Para facilitar su uso, se utiliza “push” para meter los datos.

Finalmente, se reinician las listas para prepararse para la siguiente iteración, y de esa forma se tienen ya ordenadas todas las unidades. Para la siguiente iteración, existe un iterador llamado multiplicador, el cuál marca la pauta para el caso base y también para la división de unidad, decena, centena, etc. El valor por defecto de multiplicador es 1, sin embargo, por iteración se multiplica por 10, lo que permite que, al repetir el proceso, se tomen las decenas y posteriores para ordenarlos, cumpliendo así con el objetivo del método de ordenamiento Radix sort.

El método termina cuando multiplicador llega a ser igual o mayor al número 100000, el cuál es el límite de dígitos admitidos. Al llegar a ese número, se imprime la cola original, la cuál ya se encuentra ordenada, sin embargo, se encuentra ordenada de mayor a menor, para volverla de menor a mayor, simplemente se utiliza el método reverse.

Para terminar, se realizó una función con nombre: “llamar-radix-sort” la cuál facilita su utilización, evitando que el usuario escriba el multiplicador por defecto 1.

Para utilizar el programa se deberá llamar a la función “llamas-radix-sort”, y pasarle como parámetro la lista con todos los datos que se deseen ordenar.

Análisis de complejidad:

Ejemplos uso:

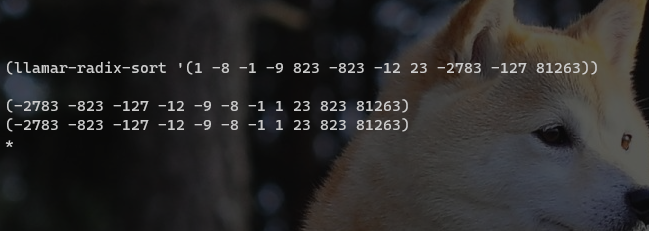
Con tal de demostrar de mejor manera el funcionamiento de esta implementación del algoritmo a common Lisp, aquí se muestran algunos ejemplos de su uso y forma de utilización, no siendo pruebas, más bien, ejemplos de cómo funciona:

  
Se carga el archivo en la terminal.

Interfaz de usuario gráfica

Descripción generada automáticamente

Se llama a la función dándole una lista de números aleatorios y como se puede observar los ordena de forma correcta, incluso a los repetidos y a los demás de un dígito.



Al probar con números negativos o de muchos dígitos que aun estén en el rango dado, se puede observar que con tan solo llamar la función y darle la lista los ordena de forma correcta.

Con estos ejemplos, podemos resumir que, para poder utilizar el programa, basta con cargarlo y escribir el siguiente comando:  
(llamar-radix-sort '(lista))

En donde lista representa la lista de números desordenados que se le dará.

Pruebas y evaluación:

Se presenta una tabla con todas las pruebas realizadas y los resultados que fueron arrojados:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | N | Insertado | Esperado | Real | ¿Qué pasó? |
| 1 | 5 | 1,2,3,4,5 | 1 -> 2 -> 3 -> 4 -> 5 |  | Los datos que ya estaban ordenados los dejó igual. |
| 2 | 5 | 5,4,3,2,1 | 1 -> 2 -> 3 -> 4 -> 5 |  | Datos ordenados de mayor a menor, los ordena de menor a mayor |
| 3 | 0 | ---- | ----- |  | Escribe que está vacío |
| 4 | 1 | 5 | 5 |  | Al agregar solo un dato solamente lo muestra |
| 5 | 4 | 1, 10, 100, 1000 | 1 -> 10 -> 100 -> 1000 |  | Diferentes datos iguales ordenados con un cero cada vez más, los deja igual |
| 6 | 4 | 1000, 100, 10, 1 | 1 -> 10 -> 100 -> 1000 |  | Diferentes datos iguales ordenados de mayor a menor con un cero cada vez más, los ordena de menor a mayor |
| 7 | 4 | 9, 89, 789, 6789 | 9 -> 89 -> 789 -> 6789 |  | Diferentes datos ordenados con un dígito cada vez más, los deja igual. |
| 8 | 4 | 6789, 789, 89, 9 | 9 -> 89 -> 789 -> 6789 |  | Diferentes datos ordenados de mayor a menor con un dígito cada vez más, los ordena de menor a mayor |
| 9 | 4 | 0, 0, 0, 0 | 0 -> 0 -> 0 -> 0 |  | Si se tienen solo ceros solamente los escribe |
| 10 | 4 | 1, 20, 300, 4000 | 1 -> 20 -> 300 -> 4000 |  |  |
| 11 | 4 | 4000, 300, 20, 1 | 1 -> 20 -> 300 -> 4000 |  |  |
| 12 | 5 | 1, 1, 1, 1, 1 | 1 -> 1 -> 1 -> 1 -> 1 |  | Si todos los datos son iguales solamente los escribe. |
| 13 | 4 | 00001, 0002, 03, 004 | 1 -> 2 -> 3 -> 4 |  |  |
| 14 | 2 | 9999, 1 | 1 -> 9999 |  | Logra ordenar 2 datos. |
| 15 | 8 | 5, 12, 8, 1, 98, 76, 6, 2 | 1 -> 2 -> 5 -> 6 -> 8 -> 12 -> 76 -> 98 |  | Varios datos aleatorios los ordenó correctamente. |
| 16 | 6 | -1, 2, -10, -78, 65, -9 | -78 -> -10 -> -9 -> -1 -> 2 -> 65 |  | Ordena correctamente números negativos y positivos. |
| 17 | 7 | -1, 2, -10, -78, 65, -9, 0 | -78 -> -10 -> -9 -> -1 -> 0 -> 2 -> 65 |  | Ordena correctamente números negativos y positivos junto con el 0 |

Calendario

Descripción generada automáticamente con confianza media

En esta imagen se ilustran las pruebas que se realizaron, las cuales dieron los resultados obtenidos en la tabla de la parte de arriba.

Gracias a los resultados obtenidos de las pruebas, puedo decir con bastante seguridad que esta implementación es completa, ya que no presenta errores en ningún caso probado, ofreciendo la respuesta esperada para cada caso, además, implementa la parte para tomar en cuenta los números negativos, ordenándolos de forma correcta según su valor en negativo.

Gracias a los procesadores de hoy en día, la tarea de ordenar datos se ha vuelto prácticamente instantánea, por lo que no se pudo probar un método para obtener el tiempo que tarda esta implementación, sin embargo, con todos los conjuntos el resultado ha sido arrojado de manera casi instantánea, lo que nos habla bien de la eficiencia de esta implementación y algoritmo.

A diferencia de otros métodos RadixSort es muy eficiente, ya que con conjuntos de datos muy extensos logra ordenarlos en poco tiempo, a diferencia de algunos que pueden llegar a tardar varios minutos o incluso horas ordenando. A pesar de todas sus ventajas, la principal desventaja de este algoritmo es que se requiere saber los dígitos del número más grande a ordenar, por lo que en algunos casos es el más eficiente y en otros conviene más utilizar un ordenador más tradicional.