Linked-List-Implemented Quick Sort Algorithm

Documentation

Created by

Dat Quoc Ngo

November 28, 2018

# Summary

Linked-List-Implemented Quick Sort Algorithm is a popular algorithm that engineers have used for decades. Most of online-available source codes relevant to this algorithm is written in Java, C++, or other advanced languages. In this project, I implemented the algorithm in MIPS.

# Linked-List-Implemented Quick Sort

Linked-List-Implemented Quick Sort Algorithm project is a collection of MIPS-written functions that are similar to ones provided by Java for Linked List interface:

* addToEnd
* addAt
* cleanList
* declareLinkedList
* deleteAt
* gemerateRandomLinkedList
* printList
* printLine
* quickSort
* addNode
* mergeList
* newList
* partition
* pickPivotAtMiddle

As this project implemented linked lists that were built based on nodes. Hence, in MIPS, every node is represented by 8 bytes. First four bytes is responsible for holding value (in this project, value is an integer) and the rest is responsible for holding the address of the next node. Therefore, to access the value of a node, type the following pattern lw $t0, 0($v0) that $v0 stores the address of the node. To access the next node, type lw $v0, 4($v0). This command is used to update pointer.

This project could be developed further to a complete advanced language by using Macros. However, right now, all codes are in MIPS that requires users to strictly follow instructions described below.

# Function Details

# addToEnd

* Add a new node to the end of the list
* How: this method receives the address of the list and moves the pointer to the end of the list. Then links the last node to the new node and set the assigned value to the new one.
* Input:
* $a0 – the address of the head node
* $a1 – the value of the node needs adding
* Output:
* $v0 – the address of the adjusted node

# addAt

* Add a new node at a specific location in the list
* How: this method receives the address of the list and moves the pointer to the requested location. Then, links the previous node to the new node and links the new node to the next node. If the location users want to add a node is out of range, an error message will be printed to the console.
* Input:
* $a0 – the address of the list
* $a1 – the location that a new node is added at
* $a2 – the value of the new node
* Output:
* $v0 – the address of the adjusted list

# addNode

* Add a new node at the given address
* How: this method set the given value to the given address. Then, generate a new address by calling a MIPS syscall (code $v0 = 9) and set this new address to the next-address holder of the current node
* Input:
* $t4 – the address of the previous node
* $t3 – the value of the node
* Output:
* $v0 – the address of the next node

# cleanList

* Clean the list
* How: Implemented the lazy clean list, change the value of the head node to 0 and set the next-address holder to 0 to erase the next node
* Input:
* $a0 – the address of the empty list
* Output: void

# declareLinkedList

* Declare a random address for the list or node
* How: this method revokes *newList* sub-function to complete the task
* Input: none
* Output:
* $v0 – the address of the new node or list

# deleteAt

* Delete a node at specific location
* How: this method receives the address of the list and moves the pointer to the node before the node needs deleting. Then, link the previous node to the node that is after the node users want to delete. This process is done by setting the address of the next node to the next-address holder of the previous node. If the location of the node that users want to delete is out of bound, an error message will be printed to the console.
* Input:
* $a0 – the address of the list
* $a1 – the location that a node needs deleting
* Output:
* $v0 – the address of the adjusted list

# generateRandomLinkedList

* Add randomly generated numbers to the list
* How: this method utilizes the MIPS sycall (code $v0 = 42) to generate random numbers. Then revoke *addNode* function (mentioned above) to add the node to the list
* Input:
* $a0 – the number of the nodes in the list
* $a1 – the address of the list or head node
* Output:
* $v0 – the address of the head node or randomly value-generated list
* $v1 – the address of the last node

# patrition

* Parse nodes into correct lists based on their values and pivot
* How: this method iteratively goes through the list and compares each node with the pivot. Smaller nodes are put into the smaller list, same nodes are put into the same list, and larger nodes are put into the larger list
* Input:
* $t0 – the address of the need-parsing list
* $t1 – the pivot
* $s0 – the address of the smaller list
* $s1 – the address of the same list
* $s2 – the address of the larger list
* Output:
* $t0 – the address of the adjusted smaller list
* $t1 – the address of the adjusted same list
* $t2 – the address of the adjusted larger list

# pickPivotAtMiddle

* Select the pivot that is the node at the center of the list
* How: this method has a slow pointer and a fast pointer that iterates through the list two times faster than the slow pointer. When the fast pointer reaches the end of the list, the slow pointer points to the node in the center of the list.
* Input:
* $a0 – the address of the list or head node
* Output:
* $v0 – the pivot value

# printList

* Print the entire list to the console
* How: this method iterates through the list and print the value of each node the console by revoking the MIPS syscall (code $v0 = 1)
* Input:
* $s5 – the address of the list or head node
* Output:
* Void

# printLine

* Print an empty line
* How: this method revokes the MIPS syscall (code $0 = 4) to print the ASCII character ‘\n’ representing a return
* Input: none
* Output: void

# mergeList

* Merge three lists provided by quickSort function
* How: this method receives the addresses of three lists (smaller, same, and larger). Then, iteratively relink nodes in the same list to the smaller list; and iteratively relink nodes in the larger list to the adjusted smaller list
* Input:
* $s0 – the address of the smaller list
* $s1 – the address of the same list
* $s2 – the address of the larger list
* $t0 – the address of the smaller list
* $t1 – the address of the same list
* $t2 – the address of the larger list
* Output:
* $s0 – the address of the merged list
* $t2 – the address of the last node in the merged list

# newList

* Declare a new list or new node
* How: this method utilizes the MIPS syscall (code $v0 = 9) to allocate random memory address to the list or node.
* Input: none
* Output: $v0 – the address of the new list or new node

# quickSort

* Perform quick-sort algorithm to the list
* How: this method implements the quick-sort algorithm described in the following website <https://www.geeksforgeeks.org/quick-sort/> . This method revokes functions defined in this project to complete the sorting process.
* Input:
* $v0 – the address of the list or the head node
* $v1 – the address of the last node in the list
* Output:
* $v0 – the address of the list or the head node
* $v1 – the address of the last node in the list

# Sample Code

main:

#create linked list

jal declareLinkedList

move $a1, $v0 #pass arguments

li $a0, 5 #set the number of nodes to $a0

jal generateRandomLinkedList

move $s5, $v0 #print out the initial list

jal printLine

jal printList

#pass arguments and do quick sorting

jal quickSort

move $s5, $v0 #print the sorted list

jal printLine

jal printList

move $a0, $v0 #delete demo

li $a1, 10

jal deleteAt

move $s5, $v0 #print just-deleted list

jal printLine

jal printList

move $a0, $v0 #add demo

li $a1, 6

li $a2, 3

jal addAt

move $a0, $v0

li $a1, 3

jal addToEnd

move $s5, $v0 #print added list

jal printLine

jal printList

Exit: li $v0, 10

syscall

# Expected Sample Output

94 55 47 6 68

6 47 55 68 94 Delete: out of range

6 47 55 68 94 Add: out of range

6 47 55 68 94 3

-- program is finished running --