Project: Sorting

Samael Newgate

University of Advancing Technology

Documentation

**Project Description:** This is a sorting project that uses an insertion sort and recursive insertion sort algorithm to arrange a doubly linked list.

**Sorting:** Through a comparison operator of elements, sorting algorithms arrange elements. There are multiple types of sorting algorithms.

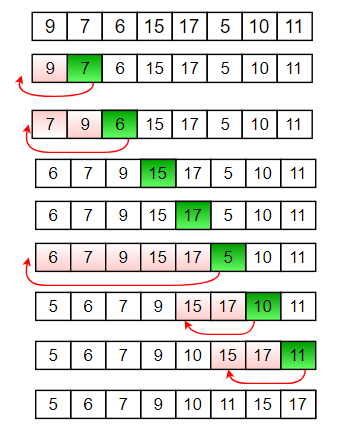
List of sorting algorithms:

|  |  |  |
| --- | --- | --- |
| Selection Sort | Bubble Sort | Recursive Bubble Sort |
| Insertion Sort | Recursive Insertion Sort | Merge Sort |
| Iterative Merge Sort | Quick Sort | Iterative Quick Sort |
| Heap Sort | Counting Sort | Radix Sort |
| Bucket Sort | Shell Sort | Tim Sort |
| Comb Sort | Cocktail Sort | Strand Sort |
| Bitonic Sort | Pancake Sort | Binary Insertion Sort |
| Gnome Sort | Bogo Sort(Permutation Sort) | Sleep Sort |
| Structure Sort | Tag Sort | Tree Sort |
| Cartesian Tree Sort | Odd-Even Sort/ Brick Sort | Quick Sort (Singly LL) |
| Quick Sort (Doubly LL) | 3-Way Quick Sort | Merge Sort (LL) |
| Merge Sort (Double LL) | 3-Way Merge Sort |  |

**Insertion Sort:** While insertion is efficient when it comes to a small number of elements because of how simple it is. It sorts one item at a time just like we would sort cards in our hands. Because of this, it is inefficient when it comes to a large number of elements.



**Recursive Insertion Sort:** Recursive insert sorting works almost like insertion sort. The only difference is, it breaks down the array to a smaller array and sorts an item at a time. For example an array for 6 elements would get broken down to an array of 1, and the 2nd node would be inserted into the sorted array. Then the array would expand for an array of 2 and the 3rd node would be inserted into the sorted array and so on until there is no node left.



**Sorting Structure:**

void insert\_sort(LinkedList<int> & List);

void recur\_sort(LinkedList<int> & List, LinkedList<int>::Node\* prev);

void display\_list(LinkedList<int> & List);

LinkedList<int>::Node\* partition(LinkedList<int> & List, LinkedList<int>::Node\* low, LinkedList<int>::Node\* high);

int main(int argc, char\* argv[])

{

LinkedList<int> SortLL;

if (argv[1] == NULL || argv[2] == NULL)

{

std::cout << "There are not enough arguments. Please enter the file to sort and the number to indicate the sorting algorithm.\n";

}

else

{

std::ifstream infile(argv[1]);

std::string line;

clock\_t start = NULL;

clock\_t duration = NULL;

if (infile.is\_open())

{

while (std::getline(infile, line))

{

SortLL.insert(std::stoi(line, nullptr, 10));

}

infile.close();

std::cout << "Sorting the list:\n";

if (\*argv[2] == '1')

{

std::cout << "Insertion Sort\n";

start = clock();

insert\_sort(SortLL);

duration = clock() - start;

}

else if (\*argv[2] == '2')

{

std::cout << "Recursive Insertion Sort\n";

start = clock();

recur\_sort(SortLL, SortLL.get\_tail());

duration = clock() - start;

}

else

{

std::cout << "ERROR: Improper command line argument value" << std::endl;

}

std::cout << "\nTime it took to sort: " << (float)duration/(CLOCKS\_PER\_SEC/1000) << " milliseconds" << std::endl;

if (SortLL.get\_size() <= 100)

{

display\_list(SortLL);

}

}

else

{

std::cout << "ERROR: File could not be opened.\n";

}

}

system("pause");

return 0;

}

void insert\_sort(LinkedList<int> & List)

{

if (List.get\_head() == nullptr)

{

return;

}

else

{

auto current = List.get\_head();

while (current->next != nullptr)

{

current = current->next;

auto sorted = current->prev;

int sort\_val = current->data;

int temp = 0;

while (sorted != nullptr && sorted->data > sort\_val)

{

temp = sorted->data;

sorted->data = sort\_val;

sorted->next->data = temp;

sorted = sorted->prev;

}

}

}

}

void recur\_sort(LinkedList<int> & List, LinkedList<int>::Node\* prev)

{

auto sorted = prev->prev;

if (prev == List.get\_head())

{

return;

}

else

{

recur\_sort(List, sorted);

int sort\_val = prev->data;

int temp = 0;

while (sorted != nullptr && sorted->data > sort\_val)

{

temp = sorted->data;

sorted->data = sort\_val;

sorted->next->data = temp;

sorted = sorted->prev;

}

}

}

// Display our linkedlist

void display\_list(LinkedList<int> & List)

{

auto current = List.get\_head();

std::cout << "List:\n";

while (current != nullptr)

{

std::cout << current->data << std::endl;

current = current->next;

}

std::cout << std::endl;

}

**Analysis:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Algorithm | 10 numbers | 100 numbers | 1,000 numbers | 10,000 NUMBERS | 100,000 NUMBERS | 1,000,000  NUMBERS |
| INSERTION SORT | **0 MS** | **0 MS** | **23 MS** | **121 MS** | **14380 MS** | **STALLED** |
| RECURSIVE INSERTION SORT | **0 MS** | **0 MS** | **22 MS** | **STACK OVERFLOW** | **STACK OVERFLOW** | **STACK OVERFLOW** |

**Usage:** Where sorting algorithms usually shine in is when it comes to sorting arrays. Deciding what sorting algorithm, you’re going to use depends on what is stored in the array, how they will be compared, what your end goal is, how big the array is, etc. For example, a quick sort would be great if you’re wanting an average case of Big O instead of a stable sort. Merge sort would work great if you want a stable sort but a Big O of O(N log N). Insertion sort would be a great option if the N value is small, meaning the element amount to be sorted is small. Each sorting algorithm has it’s uses with their own pros and cons.

**Linked List**: Linked lists are a linear data structure where the data is linked using pointers.



**Doubly Linked List**: For this project I used a doubly linked list which works almost the same as a linked list. The only difference is you have a head and a tail not just a head. You also have an extra pointer called previous pointer. This is so you can go back and forth on the list. Not just forward like on the linked list.



**Linked List Class Structure:**

template<typename T>

class LinkedList

{

// Public variables, constructor and deconstructor

public:

struct Node

{

T data;

Node\* prev;

Node\* next;

Node(T in\_data)

{

data = in\_data;

prev = nullptr;

next = nullptr;

}

};

LinkedList();

~LinkedList();

Node\* get\_head();

Node\* get\_tail();

long get\_size();

void insert(T data);

Node\* find(T data\_find);

void remove(T data\_delete);

Node\* maximum();

Node\* minimum();

// Private variables, head and tail

private:

Node\* head;

Node\* tail;

long size;

};

**Linked List Member Functions:**

|  |  |
| --- | --- |
| LinkedList() | Constructor |
| ~LinkedList() | Deconstructor |
| get\_head() | Gets the head pointer of the list(Refer back to linked list or doubly linked list diagram example on page 1) |
| get\_tail() | Gets the tail pointer of the list(Refer back to doubly linked list diagram example on page 1) |
| insert(T data) | Inserts data type T data to the linked list |
| find(T data\_find) | Returns a specific data type T data pointer from the list |
| remove(T data\_delete) | Deletes specific data type T data from the list |
| maximum() | Returns the maximum data type T value in the linked list |
| minimum() | Returns the minimum data type T value in the linked list |
| head | Head pointer |
| tail | Tail Pointer |
| struct Node | Struct for our node, this allows us to not have to use a class for node. Which allows us to be able to call less functions. |

**Usage:** Linked lists are great for organizing different types of data. Get\_head or get\_tail could be used if you’re trying to access the last or newest data that was inserted. This would depend on how you structure your linked list, whether it be inserted from the head or the tail. Find would help you look up the data you want to access. Remove can be used to get rid of data that isn’t needed anymore. The minimum and maximum have an infinite amount of uses, it depends on the type of data that was stored. If you’ve stored debts, you could use maximum to return the biggest debt. Head and tail work as boundaries for the data structure to work within.