Documentation:

- Our data structures can store addresses only.
- You are supposed to store address of any element of any data type (structure , int ,char , double , class …etc.) in our data structure , but you have to do type casting whenever you insert and extract data in our data structures.
- You are supposed to do type casting of address of data to void * type whenever you insert address of data.
- You are supposed to do type casting of data to required data type whenever you extract data from our data structure as our data structure return value of (void *) type.

header

<tm_sll.h>

Singly linked list

Singly linked list allow constant time insert and erase operations anywhere within the sequence.

tm_sll are implemented as singly-linked lists; Singly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept by the association to each element of a link to the next element in the sequence.

A singly linked list one has to iterate from the beginning to that position, which takes linear time in the distance between these. They also consume some extra memory to keep the linking information associated to each element (which may be an important factor for large lists of small-sized elements).

The tm_sll has been designed with efficiency in mind: By design, it is very efficient. We store addresses of element instead of actual data which save memory in some way, and we provide iterator feature in our tm_sll which makes its unique which decrease time complexity of traversing and searching of element.

Functions

createSinglyLinkedList

> create singly linked list
>
> **Parameter : Address of bool type element**
>
> **Return Value : (SinglyLinkedList *)**

destroySinglyLinkedList

destroy singly linked list

Parameter : Address of SinglyLinkedList

Return Value : (void)

destroySinglyLinkedList

return size of singly linked list

Parameter : Address of SinglyLinkedList

Return Value : (int)

addToSinglyLinkedList

add element to singly linked list

Parameter : Address of SinglyLinkedList ,data (type cast to void *) and bool type element

Return Value : (void)

insertIntoSinglyLinkedList

insert element at given position

Parameter : Address of SinglyLinkedList ,index(int),address of data (type cast to void *) and address of bool type element

Return Value : (void)

removeFromSinglyLinkedList

remove element from list

Parameter : Address of SinglyLinkedList,index(int) and address of bool type element

Return Value : (void *)

clearSinglyLinkedList

clear content of singly linked list

Parameter : Address of SinglyLinkedList

Return Value : (void)

appendToSinglyLinkedList

append two list

Parameter : Address of SinglyLinkedList , address of SinglyLinekdList and address of bool type element

Return Value : (void)

getFromSinglyLinkedList

return element of given position

Parameter : Address of SinglyLinkedList , index(int) and address of bool type element

Return Value : (void *)

getSinglyLinkedListIterator

return Iterator

Parameter : Address of SinglyLinkedList and bool type element

Return Value : (SinglyLinkedListIterator)

hasNextInSinglyLinkedList

test whether list has next element or not

Parameter : Address of SinglyLinkedListIterator

Return Value : (bool)

getNextElementFromSinglyLinkedList

return next element of list

Parameter : Address of SinglyLinkedListIterator and bool type element.

Return Value : (void *)

Library properties

Sequence

Elements in sequence :Elements are ordered in a strict linear sequence. Individual elements are accessed by their position in this sequence.

Linked list

Each element keeps information on how to locate the next element, allowing constant time insert and erase operations after a specific element (even of entire ranges), but no direct random access.

Example :

```
#include<tm_sll.h>
#include<string.h>
#include<stdio.h>
typedef struct student{
char name[21];
int rollNo;
char gender[10];
}std;
int main(){
bool success;
SinglyLinkedList *list ;
std s1,s2,s3,*s;
SinglyLinkedListIterator iter;
list = createSinglyLinkedList(&success);
strcpy(s1.name,"Vikas");
s1.rollNo=101;
strcpy(s1.gender,"Male");
addToSinglyLinkedList(list,(void *)&s1,&success);
if(success==false)
{
printf("Unable to add\n");
```

```
}
strcpy(s2.name,"Tanish");
s2.rollNo=102;
strcpy(s2.gender,"Male");
addToSinglyLinkedList(list,(void *)&s2,&success);
strcpy(s3.name,"Benny");
s3.rollNo=103;
strcpy(s3.gender,"Female");
addToSinglyLinkedList(list,(void *)&s3,&success);
printf("Traversing technique one\n");
for(int i=0;i<getSizeOfSinglyLinkedList(list);i++)
{
s=(std *)getFromSinglyLinkedList(list,i,&success);
printf("%s,%s,%d\n",s->name,s->gender,s->rollNo);
}
printf("Traversing technique two\n");
iter=getSinglyLinkedListIterator(list,&success);
if(success)
{
while(hasNextInSinglyLinkedList(&iter))
{
s=(std *)getNextElementFromSinglyLinkedList(&iter,&success);
printf("%s,%s,%d\n",s->name,s->gender,s->rollNo);
}
}
printf("Size of singly linked list is %d\n",getSizeOfSinglyLinkedList(list));
clearSinglyLinkedList(list);
destroySinglyLinkedList(list);
return 0;
}
```

<tm_queue.h>

queue

FIFO queue

Queue are a type of ds, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end and extracted from the other.

Elements are *pushed* into the *"back"* of the specific queue and *popped* from its *"front"*.

Functions

createQueue
: create queue
Parameter : Address of bool type element
Return Value : (Queue *)

addToQueue
: add element at queue
Parameter : Address of queue ,data (type cast to void *) and bool type element
Return Value : (void)

removeFromQueue
: remove element from queue
Parameter : Address of queue and bool type element
Return Value : (void *)

getSizeOfQueue
: return size
Parameter : Address of queue
Return Value : (int)

elementAtFrontOfQueue
: remove element
Parameter : Address of queue and bool type element
Return Value : (void *)

isQueueEmpty
: check whether queue is empty or not
Parameter : Address of queue
Return Value : (bool)

clearQueue
: clear contents of queue
Parameter : Address of queue
Return Value : (void)

destroyQueue
: destroy queue
Parameter : Address of queue
Return Value : (void)

Example :

```
#include<tm_queue.h>
#include<stdio.h>
int main()
{
int succ;
int X1,X2,X3,X4;
int *x;
X1=10;
X2=20;
X3=30;
X4=40;
Queue *queue;
queue=createQueue(&succ);
if(queue==false)
{
printf("unable to create queue \n"); return 0;
}
addToQueue(queue,&X1,&succ);
if(succ) printf("%d added to queue \n", X1);
else printf("unable to add %d to queue \n" ,X1);

addToQueue(queue,&X2,&succ);
if(succ) printf("%d added to queue \n", X2);
else printf("unable to add %d to queue \n" ,X2);

addToQueue(queue,&X3,&succ);
if(succ) printf("%d added to queue \n", X3);
else printf("unable to add %d to queue \n" ,X3);

addToQueue(queue,&X4,&succ);
```

```
if(succ) printf("%d added to queue \n", X4);
else printf("unable to add %d to queue \n" ,X4);

printf("size of queue %d \n ",getSizeOfQueue(queue));
if(isQueueEmpty(queue)) printf("Queue is empty \n");
else printf("Queue is not empty \n");
x=(int *)elementAtFrontOfQueue(queue,&succ);
printf("Element at front of queue is %d \n", *x);
printf("Now removing all element from queue \n");
while(!isQueueEmpty(queue))
{
x=(int*)removeFromQueue(queue,&succ);
printf("%d removed from queue \n ",*x);
}
destroyQueue(queue);
return 0;
}
```

<tm_stack.h>

stack

Stacks in Data Structures is a linear type of data structure that follows the LIFO (Last-In-First-Out) principle and allows insertion and deletion operations from one end of the stack data structure, that is top.

Functions

createStack
: create stack
Parameter : Address of bool type element
Return Value : (Stack *)

getSizeOfStack
: return size
Parameter : Address of stack
Return Value : int

popFromStack

> pop element from stack
> **Parameter : Address of stack and bool type element**
> **Return Value : (void *)**

pushOnStack

> push element on stack
> **Parameter : Address of stack ,data (type cast to void *) and bool type element**
> **Return Value : (Stack *)**

elementAtTopOfStack

> return element at top of stack
> **Parameter : Address of stack and bool type element**
> **Return Value : (void *)**

isStackEmpty

> check whether stack is empty or not
> **Parameter : Address of stack**
> **Return Value : bool**

clearQueue

> clear contents of stack
> **Parameter : Address of stack**
> **Return Value : (void)**

destroyQueue

> destroy stack
> **Parameter : Address of stack**
> **Return Value : (void)**

Example:

```
#include<tm_stack.h>

#include<stdio.h>

int main()

{

int succ;

int X1,X2,X3,X4;

int *x;

X1=10;

X2=20;

X3=30;

X4=40;

Stack *stack;

stack=createStack(&succ);

if(stack==false)

{
```

```
printf("unable to create stack \n"); return 0;
}
pushOnStack(stack,&X1,&succ);
if(succ) printf("%d pushed to stack \n", X1);
else printf("unable to push %d to stack \n" ,X1);

pushOnStack(stack,&X2,&succ);
if(succ) printf("%d pushed to stack \n", X2);
else printf("unable to push %d to stack \n" ,X2);

pushOnStack(stack,&X3,&succ);
if(succ) printf("%d pushed to stack \n", X3);
else printf("unable to push %d to stack \n" ,X3);

pushOnStack(stack,&X4,&succ);
if(succ) printf("%d pushed to stack \n", X4);
else printf("unable to push %d to stack \n" ,X4);

printf("size of stack %d \n ",getSizeOfStack(stack));
if(isStackEmpty(stack)) printf("Stack is empty \n");
else printf("Stack is not empty \n");
x=(int *)elementAtTopOfStack(stack,&succ);
printf("Element at top of stack is %d \n", *x);
printf("Now removing all element from stack \n");
while(!isStackEmpty(stack))
{
x=(int *)popFromStack(stack,&succ);
printf("%d popped from stack \n ",*x);
}
destroyStack(stack);
return 0;
}
```

<tm_sort.h>

Linear Sort:

- Linear sorting algorithms,have a time complexity of O(n), making them highly efficient for specific scenarios.
- They are non-comparative sorting methods and are efficient for specific types of data.

Bubble Sort:

- Bubble sort is a simple sorting algorithm that repeatedly compares and swaps adjacent elements if they are in the wrong order.
- It has a worst-case time complexity of O(n^2), making it inefficient for large datasets.
- Bubble sort is easy to understand and implement but is generally not used for sorting large or complex lists.

Selection Sort:

- Selection sort divides the list into a sorted and an unsorted part.
- It repeatedly selects the minimum (or maximum) element from the unsorted part and moves it to the sorted part.
- Selection sort has a time complexity of O(n^2), which can make it inefficient for large lists.

Insertion Sort:

- Insertion sort builds the sorted list one element at a time by comparing and inserting elements in their correct positions.
- It is efficient for small datasets and lists that are already partially sorted.
- The average and worst-case time complexity of insertion sort is O(n^2).

Merge Sort:

- Merge sort is a divide-and-conquer algorithm that divides the list into smaller sublists, sorts them, and then merges them into a sorted list.
- It is a stable sorting algorithm with a time complexity of O(n log n), making it efficient for large datasets.
- Merge sort is commonly used in practice and is suitable for various data types.

Quick Sort:

- Quick sort is another divide-and-conquer algorithm that works by selecting a "pivot" element and partitioning the list into elements less than and greater than the pivot.
- It has an average time complexity of O(n log n), making it one of the fastest sorting algorithms.
- However, its worst-case time complexity can be O(n^2) if not implemented properly, although this is rare in practice.
- Quick sort is widely used and is often the default sorting algorithm in many programming languages and libraries.

These sorting algorithms vary in terms of efficiency, ease of implementation, and suitability for different scenarios, so the choice of which one to use depends on the specific requirements of a given task.

Functions

bubbleSort
: use bubble sort algorithm for sorting
Parameter : Address of data (type cast to void *) , collection size(int) , element size (int) and Comparator (return type:int , parameter: (void *),(void *))
Return Value : (void)

linearSort
: use linaer sort algorithm for sorting
Parameter : Address of data (type cast to void *) , collection size(int) , element size (int) and Comparator (return type:int , parameter: (void *),(void *))
Return Value : (void)

quickSort
: use quick sort algorithm for sorting
Parameter : Address of data (type cast to void *) , lower bound(int) , upperBound(int) , element size (int) and Comparator (return type:int , parameter: (void *),(void *))
Return Value : (void)

selectionSort
: use selection sort algorithm for sorting
Parameter : Address of data (type cast to void *) , collection size(int) , element size (int) and Comparator (return type:int , parameter: (void *),(void *))
Return Value : (void)

linearSinsertionSort
: use insertion sort algorithm for sorting
Parameter : Address of data (type cast to void *) , collection size(int) , element size (int) and Comparator (return type:int , parameter: (void *),(void *))
Return Value : (void)

mergeSort

use merge sort algorithm for sorting

Parameter : Address of data (type cast to void *) , lower bound(int) , upperBound(int) , element size (int) and Comparator (return type:int , parameter: (void *),(void *))

Return Value : (void)

Example:

```
#include<stdio.h>

#include<tm_sort.h>

#include<stdlib.h>

#include<string.h>

typedef struct Student

{

int rollNumber;

char name[21];

}std;

int StudentComparator(void *left,void *right)

{

std*s1,*s2;

s1=(std*)left;

s2=(std *)right;

return s1->rollNumber-s2->rollNumber;

}

int main()

{

int req;

std *s,*j;

int y;

printf("Enter requirement : ");

scanf("%d",&req);

if(req<=0)

{

printf("Invalid requirement\n");

return 0;
```

```
}
s=(std *)malloc(sizeof(std)*req);
j=s;
for(y=0;y<req;y++)
{
printf("Enter roll number : ");
scanf("%d",&(j->rollNumber));
printf("Enter name : ");
scanf("%s",j->name);
j++;
}
printf("Before sorting records are arranged as following\n");
for(y=0;y<req;y++)
{
printf("Roll number %d , Name %s\n",s[y].rollNumber,s[y].name);
}
// bubbleSort(s,req,sizeof(std),StudentComparator);
// linearSort(s,req,sizeof(std),StudentComparator);
// selectionSort(s,req,sizeof(std),StudentComparator);
// insertionSort(s,req,sizeof(std),StudentComparator);
// mergeSort(s,0,req-1,,sizeof(std),StudentComparator);
quickSort(s,0,req-1,sizeof(std),StudentComparator);
printf("After sorting records are arranged as following\n");
for(y=0;y<req;y++)
{
printf("Roll number %d , Name %s\n",s[y].rollNumber,s[y].name);
}
free(s);
return 0;
}
```