Worksheet 7

Overview

The w orkshop for w eek 8 w ill start w ith a tutorial on Mergesort, Master Theorem and Recurrences. You w ill implement Mergesort.

Tutorial Questions

Question 7.1 Trace the action of bottom-up mergesort on the same keys as in last w eek's w orkshop:

- 2 3 97 23 15 21 4 23 29 37 5 23

Is mergesort stable?

Programming exercises

Programming 7.1 Write code for bottom-up mergesort w here the data are contained in an initially unsorted *linked list*. You w ill have to construct an artificial linked list to test your code. You can populate your linked list w ith random numbers before sorting.

```
/* ll.h */
struct linkedList {
    struct linkedList *next;
    int item;
};

/* Add a linked list item before the head of the given item. */
struct linkedList *prepend(struct linkedList *head, int item);

/* Add a linked list item to the end of the list, traversing all items. */
struct linkedList *append(struct linkedList *head, int item);

/* ll.c */
#include <stdlib.h>
#include <assert.h>
/* #include "ll.h" */

struct linkedList *prepend(struct linkedList *head, int item){
    /* FILL IN if using. */
    return NULL;
}

/* Add a linked list item to the end of the list, traversing all items. */
struct linkedList *append(struct linkedList *head, int item){
    /* FILL IN if using. */
    return NULL;
}

/* queue.h */
/* #include "ll.h" */
struct llq;

/* A simple function that appends the given list to the end
    of the queue. If the given queue is NULL, one is allocated
    with the queueHead being set to the given list, otherwise
    the existing queue is returned appended with the new item. */
struct llq *add(struct llq *oldQ, struct linkedList *list);

/* Removes head of queue from queue and returns it. Queue will
    be set to NULL if it becomes empty. */
struct linkedList *dequeue(struct llq **queue);

/* queue.c */
```

```
#include <stdlib.h>
#include <assert.h>
/* #include "ll.h" */

struct llq {
    struct linkedList *queueHead;
    struct llq *next;
};

struct llq *add(struct llq *oldQ, struct linkedList *list){
    if(! oldQ){
        oldQ = (struct llq *) malloc(sizeof(struct llq));
        assert(oldQ);
        oldQ->queueHead = list;
        oldQ->next = NULL;
    } else {
        oldQ->next = add(oldQ->next, list);
    }
    return oldQ;
}

struct linkedList *dequeue(struct llq **queue){
    struct linkedList *res = (*queue)->queueHead;
    struct llq *oldQ = *queue;

    *queue = (*queue)->next;
    /* Clean up memory we allocated. */
    free(oldQ);

    return res;
}

/* merge.h */
/* #include "ll.h" */
/* Bottom-up sorts the given list, the original list may
    be mangled in any way (and most likely will). */
struct linkedList *mergeSort(struct linkedList *list);

/* merge.c */
/* #include "ll.h" */
/* #include "merge.h" */
struct linkedList *mergeSort(struct linkedList *list){
    /* FILL IN SETUP FOR Linked List BOTTOM-UP mergesort */

    /* FILL IN loop */
}

/* main.c */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

/* 8 is a power of 2, so should be the least trouble. */
/* Try 7 to have to deal with an empty list */
#define SIZE 8

int main(int argc, char **argv){
    int inputs[SIZE];
    int i;

    struct linkedList *list = NULL;
    struct linkedList *current;

    /* Initialise random number generator. */
    time_t t;
    srand((unsigned) time(&t));

    printf("Original Array: ");
    for(i = 0; i < SIZE; i++){
        /* values between 0 and 99 are easier to read. */
        inputs[i] = rand() % 100;
        printf("%d ",inputs[i]);
    }
    printf("\n");

    printf("Adding all inputs to linked list.\n");
    for(i = 0; i < SIZE; i++){
        list = prepend(list, inputs[i]);
    }

    printf("Mergesorting.\n");
    list = mergeSort(list);
    printf("MergeSorted list: ");
    current = list;
    while(current){
        printf("%d ",current->item);
        current = current->next;
    }
    printf("\n");

    return 0;
}
```

Programming 7.2 Write code for bottom-up mergesort where the data are contained in an initially unsorted *array*. You will have to construct an artificial array to test your code. You can populate your array with random numbers before sorting.

```
/* merge.h */
/* Bottom-up mergeSorts the given list. */
int *mergeSort(int *list, int size);

/* merge.c */
#include <stdlib.h>
#include <assert.h>
int *mergeSort(int *list, int size){
    /* FILL IN loop */
}

/* main.c */
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
/* #include "merge.h" */

/* 8 is a power of 2, so should be the least trouble. */
/* Try 7 to have to deal with an empty list */
#define SIZE 8

int main(int argc, char **argv){
    int list[SIZE];
    int *sortedList;
    int i;

    /* Initialise random number generator. */
    time_t t;
    srand((unsigned) time(&t));

    printf("Original Array: ");
    for(i = 0; i < SIZE; i++){
        /* values between 0 and 99 are easier to read. */
        list[i] = rand() % 100;
        printf("%d ",list[i]);
    }

    printf("\n");

    printf("Mergesorting.\n");
    sortedList = mergeSort(list, SIZE);
    printf("MergeSorted list: ");
    for(i = 0; i < SIZE; i++){
        printf("%d ",sortedList[i]);
    }
    printf("\n");

    return 0;
```

```
  }
```