RECURSIVE VS ITERATIVE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
typedef struct node * NODE;
struct node{
    int ele;
    NODE next;
};
typedef struct linked_list * LIST;
struct linked_list{
    int count;
    NODE head;
};
LIST createNewList()
{
    LIST myList;
    myList = (LIST) malloc(sizeof(struct linked_list));
    myList->count=0;
    myList->head=NULL;
    return myList;
}
NODE createNewNode(int value)
{
    NODE myNode;
    myNode = (NODE) malloc(sizeof(struct node));
    myNode->ele=value;
    myNode->next=NULL;
    return myNode;
}
int llSumNTR(NODE head)
{
    if (head == NULL)
        return 0;
    return head->ele + llSumNTR(head->next);
}
int llSumNTRWrapper(LIST list)
{
    return llSumNTR(list->head);
}
int llSumTR(NODE head, int sum)
{
    if (head == NULL)
        return sum;
    return llSumTR(head->next, sum + head->ele);
}
int llSumTRWrapper(LIST list)
{
    return llSumTR(list->head, 0);
}
int llSumIter(NODE head)
{
    int sum = 0;
```

```
    NODE temp = head;
    while (temp->next != NULL)
    {
        sum += temp->ele;
        temp = temp->next;
    }
    return sum;
}
int llSumIterWrapper(LIST list)
{
    return llSumIter(list->head);
}
int main(int argc, char** argv)
{
    char *filename = (char *) malloc(sizeof(char)*strlen(argv[1]));
    filename = argv[1];

    char substring[strlen(filename) - 7];
    strncpy(substring, argv[1] + 7, strlen(argv[1]) - 3);

    int n = atoi(substring);

    FILE *fp;
    fp = fopen(filename, "r");
    if (fp == NULL)
    {
        printf("Error opening file\n");
        exit(1);
    }
    int num;
    LIST myList = createNewList();
    while (fscanf(fp, "%d", &num) != EOF)
    {
        NODE myNode = createNewNode(num);
        myNode->next = myList->head;
        myList->head = myNode;
        myList->count++;
    }
    fclose(fp);
    struct timeval t1, t2;
    double time_taken;

    gettimeofday(&t1, NULL);
    llSumNTRWrapper(myList);
    gettimeofday(&t2, NULL);
    time_taken = (t2.tv_sec - t1.tv_sec) * 1e6;
    time_taken = (time_taken + (t2.tv_usec - t1.tv_usec)) * 1e-6;
    printf("Non-tail recursive llsum finding took %f seconds to execute\n", time_taken);
    gettimeofday(&t1, NULL);
    llSumTRWrapper(myList);
    gettimeofday(&t2, NULL);
    time_taken = (t2.tv_sec - t1.tv_sec) * 1e6;
    time_taken = (time_taken + (t2.tv_usec - t1.tv_usec)) * 1e-6;
    printf("Tail recursive llsum took %f seconds to execute\n", time_taken);
```

```
    gettimeofday(&t1, NULL);
    llSumIterWrapper(myList);
    gettimeofday(&t2, NULL);
    time_taken = (t2.tv_sec - t1.tv_sec) * 1e6;
    time_taken = (time_taken + (t2.tv_usec - t1.tv_usec)) * 1e-6;
    printf("Iterative llsum took %f seconds to execute\n", time_taken);
    return 0;
}
```

LOMUTO PARTITIONING

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void swap(int arr[], int low, int high)
{
    // arr[low] = arr[high] ^ arr[low];
    // arr[high] = arr[high] ^ arr[low];
    // arr[low] = arr[low] ^ arr[high];
    int temp = arr[low];
    arr[low] = arr[high];
    arr[high] = temp;
}
// Ls[lo..hi] is the input array; Ls[pInd] is the pivot
int part(int Ls[], int lo, int hi, int pInd)
{
    int i = lo - 1;
    int j = lo;
    int pivot = Ls[pInd];
    swap(Ls, pInd, hi-1);
    for (; j < hi - 1; j++)
    {
        if (Ls[j] <= pivot)
        {
            i++;
            swap(Ls, i, j);
        }
    }
    swap(Ls, i+1, hi - 1);
    return i+1;
}
void printArray(int arr[], int n)
{
    printf("The array is : \n");
    for (int i = 0; i < n; i++)
    {
      printf(" %d", arr[i]);
    }
    printf("\n");
}
int main()
{
    srand(time(NULL));
    int *arr = (int*) malloc(sizeof(int)*10);
    for (int i = 0; i < 10; i++)
    {
```

```
        if (i%2 == 0) arr[i] = rand() % 100;
        else arr[i] = rand() % 50;
    }
    printArray(arr, 10);
    int pivot = rand() % 10;
    int pivotEle = arr[pivot];
    int pivotIndex = part(arr, 0, 10, pivot);
    printArray(arr, 10);
    printf("\nThe pivot index is %d for the pivot %d\n", pivotIndex, pivotEle);
    return 0;
}
```

HOARE PARTITIONING

```
// THREE WAY – CALL int pivotIndex = threePart(arr, 0, 10, pivot); for array of length 10
int threePart(int Ls[], int lo, int hi, int pInd)
{
    swap(Ls, pInd, hi - 1);
    int pivPos, lt, rt, mid, pv;
    lt = lo;
    rt = hi - 2;
    mid = lo;
    pv = Ls[hi - 1];
    while (mid <= rt)
    {
        if (Ls[mid] < pv)
        {
            swap(Ls, lt, mid);
            lt++;
            mid++;
        }
        else if (Ls[mid] > pv)
        {
            swap(Ls, mid, rt);
            rt--;
        }
        else
        {
            mid++;
        }
    }
    swap(Ls, mid, hi - 1);
    return mid;
}
// Ls[lo..hi] is the input array; Ls[pInd] is the pivot
// TWO WAY – CALL int pivotIndex = part(arr, 0, 9, pivot); for array of length 10
int part(int Ls[], int lo, int hi, int pInd)
{
    swap(Ls, pInd, lo);
    int pivPos, lt, rt, pv;
    lt = lo + 1;
    rt = hi;
    pv = Ls[lo];
    while (lt < rt)
    {
        for (; lt <= hi && Ls[lt] <= pv; lt++);
```

```
        // Ls[j]<=pv  for j in lo..lt-1
        for (; Ls[rt] > pv; rt--);
        // Ls[j]>pv  for j in rt+1..hi
        if (lt < rt)
        {
            swap(Ls, lt, rt);
            lt++;
            rt--;
        }
    }
    if (Ls[lt] < pv && lt <= hi)
        pivPos = lt;
    else
        pivPos = lt - 1;
    swap(Ls, lo, pivPos);
    // Postcond.: (Ls[j]<=pv for j in lo..pivPos-1) and (Ls[j]>pv for j in pivPos+1..hi)
    return pivPos;
}
```

RANDOM PIVOT

```
int pivot(int L[], int lo, int hi)
{
    srand(time(NULL));
    int r = rand() % (hi - lo) + lo;
    return r;
}
```

MEDIAN OF THREE PIVOT

```
int pivot(int Ls[], int lo, int hi)
{
    int left = lo;
    int center = (lo + hi)/2;
    int right = hi - 1;

    // finding the median
    if (Ls[left] >= Ls[right] && Ls[left] >= Ls[center])
    {
        if (Ls[right] >= Ls[center]) return right;
        else return center;
    }
    else if (Ls[right] >= Ls[left] && Ls[right] >= Ls[center])
    {
        if (Ls[left] >= Ls[center]) return left;
        else return center;
    }
    else if (Ls[center] >= Ls[right] && Ls[center] >= Ls[left])
    {
        if (Ls[right] >= Ls[left]) return right;
        else return left;
    }

    return center; // random
}
```

MEDIAN OF MEDIAN/QUICKSELECT PIVOT FUNCTON

```
int pivot(int L[], int lo, int hi)
```

```
{
    int n = hi - lo + 1;
    int k = n / 2; // Choose the median as the pivot
    int pivotEle = qselect(L + lo, n, k);
    for (int i = lo; i <= hi; i++)
    {
        if (L[i] == pivotEle) return i;
    }
    return lo + qselect(L + lo, n, k) - 1;
}
```

QUICKSELECT ITERATIVE

```
int qselectItr(int L[], int n, int k)
{
    int pivot = 0;
    int lo = 0;
    int hi = n - 1;

    int pInd = part(L, lo, hi, pivot);

    while (lo <= hi) {
        int pInd = part(L, lo, hi, lo); // Choose pivot as the leftmost element
        int lenLeft = pInd - lo + 1;

        if (k == lenLeft) {
            return L[pInd];
        } else if (k < lenLeft) {
            hi = pInd - 1; // Update the upper bound for the left subarray
        } else {
            k -= lenLeft; // Adjust k for the right subarray
            lo = pInd + 1; // Update the lower bound for the right subarray
        }
    }

    return -1; // If k is out of bounds
}
```

QUICK SORT

```
// CALL qs(arr, 0, n); - where n is the length of the array
void qs(int Ls[], int lo, int hi)
{
    if (lo < hi)
    {
        int p = pivot(Ls, lo, hi); // Ls[p] is the pivot
        // printf("The pivot element is %d\n", p);
        p = part(Ls, lo, hi, p); // Ls[p] is the pivot
        /*
        (Ls[j]<=Ls[p] for j in lo..pPos-1) and
        (Ls[j]>Ls[p] for j in pPos+1..hi)
        */
        qs(Ls, lo, p);
        qs(Ls, p + 1, hi);
    }
}
```

HYBRID QUICK SORT 1

```
void insertionSort(int A[], int lo, int hi)
{
   int n = hi - lo;
   for (int j = lo + 1; j < n; j++)
   {
      int i = j - 1;
      int v = A[j];
      while (i >= lo && v < A[i])
      {
         A[i+1] = A[i];
         i--;
      }
      A[i+1] = v;
   }
}
// hybrid version - insertion sort used of array size less than 10
void qs(int Ls[], int lo, int hi)
{
   if (hi - lo < 10)
   {
      insertionSort(Ls, lo, hi);
      return;
   }
   else if (lo < hi)
   {
      int p = pivot(Ls, lo, hi);
      p = part(Ls, lo, hi, p);
      qs(Ls, lo, p);
      qs(Ls, p + 1, hi);
   }
}
```

ITERATIVE QUICK SORT 1

```
// partially iterative quick sort
void qs(int Ls[], int lo, int hi)
{
  while (lo < hi)
   {
      int pInd = pivot(Ls, lo, hi);
      int p = part(Ls, lo, hi, pInd);
      qs(Ls, lo, p - 1);
      lo = p + 1;
   }
}
```

ITERATIVE QUICK SORT 2

```
void qs(int Ls[], int lo, int hi)
{
   Stack *s = newStack();
   Element ele = {lo, hi};
   push(s, ele);
   while (!isEmpty(s))
   {
      Element e = *top(s);
      pop(s);
      lo = e.lo;
```

```
        hi = e.hi;

        while (lo < hi)
        {
            int p = pivot(Ls, lo, hi);
            p = part(Ls, lo, hi, p);
            push(s, (Element){lo, p - 1});
            lo = p + 1;
        }
    }
}
```

HYBRID QUICK SORT 2

```
// hybrid version - insertion sort used at the end on the partially sorted array
void qs(int Ls[], int lo, int hi)
{
    if (hi - lo < 10)
    {
        return;
    }
    else if (lo < hi)
    {
        int p = pivot(Ls, lo, hi);
        p = part(Ls, lo, hi, p);
        qs(Ls, lo, p);
        qs(Ls, p + 1, hi);
    }

    insertionSort(Ls, 0, hi);
}
```

COUNTING SORT ON CHAR

```
char* counting_sort(char* A, char* B, int k, int n)
{
    // Initialize array C with all 0s
    int C[k];
    for (int i = 0; i < k; i++)
    {
        C[i] = 0;
    }
    // Count the number of times each element occurs in A and store it in C
    for (int j = 0; j < n; j++)
    {
        C[A[j]-97]++;
    }
    // Place the elements of A in B in the correct position
    for (int i = 1; i < k; i++)
    {
        C[i] = C[i] + C[i - 1];
    }
    for (int j = n - 1; j >= 0; j--)
    {
        B[C[A[j]-97] - 1] = A[j];
        C[A[j]-97]--;
    }
    return B;
```

```
}
int main()
{
    int n;
    printf("Enter the number of elements in the string \n");
    scanf("%d", &n);
    char *str = (char*) malloc(sizeof(char)*n);
    printf("Enter the string \n");
    scanf("%s", str);
    int k = str[0];
    for (int i = 1; i < n; i++)
    {
        if (str[i] > k)
        {
            k = str[i];
        }
    }
    k = k - 97;

    char *B = (char*) malloc(sizeof(char)*n);
    counting_sort(str, B, k + 1, n);
    printf("The sorted array is: ");
    printf("%s \n", B);
    return 0;
}
```

LAB 7 TASK 2

```
int main()
{
    FILE *fptr = fopen("n_integers.txt", "r");
    int length;
    printf("Enter the number of arrays you want to test\n");
    scanf("%d", &length);
    for (int j = 0; j < length; j++)
    {
        int n;
        char *num = (char *) malloc(sizeof(int));
        fscanf(fptr, "%[^,],[", num);
        char *line = (char*) malloc(sizeof(int));
        n = atoi(num);

        int *arr = (int*) calloc(n, sizeof(int));

        for (int i = 0; i < n-1; i++){

            fscanf(fptr, "%s", line);
            arr[i] = atoi(line);

        }
        fscanf(fptr, "%s]\n", line);
        arr[n-1] = atoi(line);

        int k = arr[0];
        for (int i = 1; i < n; i++)
        {
```

```
        if (arr[i] > k)
        {
            k = arr[i];
        }
    }
    // Initialize array C with all 0s
    int C[k];
    for (int i = 0; i < k; i++)
    {
        C[i] = 0;
    }
    // Count the number of times each element occurs in A and store it in C
    for (int j = 0; j < n; j++)
    {
        C[arr[j]]++;
    }
    // Place the elements of A in B in the correct position
    for (int i = 1; i < k; i++)
    {
        C[i] = C[i] + C[i - 1];
    }
    int a,b,target;

    printf("Enter the integer and target range it should lie in\n");
    scanf("%d %d %d", &target, &a, &b);

    if (target < C[b] && target > C[a-1])
    {
        printf("Target is present in given range ! \n");
    }
    else
    {
        printf("Target not found in desired range. \n");
    }
  }

  return 0;
}
```

STRAIGHT RADIX

```
void modified_counting_sort(int* A, int k, int n, int place)
{
    int *B = (int*) malloc(sizeof(int)*n);
    // Initialize array C with all 0s
    int C[k];
    for (int i = 0; i < k; i++)
    {
        C[i] = 0;
    }
    // Count the number of times each element occurs in A and store it in C
    for (int j = 0; j < n; j++)
    {
        C[(A[j]%(place*10))/place]++;
    }
    // Place the elements of A in B in the correct position
```

```
    for (int i = 1; i < k; i++)
    {
        C[i] = C[i] + C[i - 1];
    }
    for (int j = n - 1; j >= 0; j--)
    {
        B[C[(A[j]%(place*10))/place] - 1] = A[j];
        C[(A[j]%(place*10))/place]--;
    }
    // Copy sorted elements from B back to A
    for (int i = 0; i < n; i++)
    {
        A[i] = B[i];
    }
    free(B);
}
void radix(int* A, int n)
{
    int max = -1;
    for (int i = 0; i < n; i++)
    {
        if (A[i] > max) max = A[i];
    }
    int places = 1;
    while (max > 0)
    {
        places *= 10;
        max = max/10;
    }
    for (int i = 1; i < places; i *= 10)
    {
        modified_counting_sort(A, 10, n, i);
    }
}
```

EXCHANGE RADIX

```
void swap(int arr[], int low, int high)
{
    int temp = arr[low];
    arr[low] = arr[high];
    arr[high] = temp;
}
void convert_to_binary(int *A, int n)
{
    for (int i = 0; i < n; i++)
    {
        int m = A[i];
        int bin = 0;
        int digit = 1;

        while (m > 0)
        {
            int rem = m%2;
            bin += rem * digit;
            m = m/2;
```

```
            digit *= 10;
        }
        A[i] = bin;
    }
}
void convert_from_binary(int *A, int n)
{
    for (int i = 0; i < n; i++)
    {
        int m = A[i];
        int dec = 0;
        int digit = 1;

        while (m > 0)
        {
            int rem = m%10;
            dec += rem * digit;
            m = m/10;
            digit *= 2;
        }
        A[i] = dec;
    }
}
void printArray(int *A, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
void radix_exchange_sort_helper(int Ls[], int lo, int hi, int place, int places)
{
    if (place > 0)
    {
        int i = lo;
        int j = hi - 1;
        while (i <= j)
        {
            int down = (Ls[i]%(place*10))/place;
            int up = (Ls[j]%(place*10))/place;
            if (up == 0 && down == 1)
            {
                swap(Ls, i, j);
                i++;
                j--;
            }
            else if (down == 0) i++;
            else if (up == 1) j--;
        }
        // printArray(Ls, hi);
        radix_exchange_sort_helper(Ls, lo, i, place/10, places);
        radix_exchange_sort_helper(Ls, i, hi, place/10, places);
    }
    else
```

```
    {
        return;
    }
}
void radix_exchange_sort(int* A, int n)
{
    convert_to_binary(A, n);
    int max = -1;
    for (int i = 0; i < n; i++)
    {
        if (A[i] > max) max = A[i];
    }

    int places = 1;
    while (max > 0)
    {
        places *= 10;
        max = max/10;
    }
    // printf("Places = %d\n", places);
    radix_exchange_sort_helper(A, 0, n, places/10, places);
    convert_from_binary(A, n);
}
int main(int argc, char** argv)
{
    // int arr[7] = {7, 1, 6, 2, 5, 3, 4};
    // printArray(arr, 7);
    // radix_exchange_sort(arr, 7);
    // printArray(arr, 7);
    // testing with a fixed length and random elements
    // srand(time(NULL));
    // int n = 10;
    // int *arr = (int*) malloc(sizeof(int)*n);
    // for (int i = 0; i < n; i++)
    // {
    //    if (i%2 == 0) arr[i] = rand() % 30;
    //    else arr[i] = rand() % 20;
    // }
    // printArray(arr, n);
    // radix_exchange_sort(arr, n);
    // printArray(arr, n);
    printf("Enter the number of arrays to test \n");
    int length;
    scanf("%d", &length);
    FILE *fptr = fopen("n_integers.txt" , "r");
    if (fptr == NULL){
        printf("Error opening file");
        exit(1);
    }
    for (int j = 0; j < length; j++)
    {
        int n;
        char *num = (char *) malloc(sizeof(int));
        fscanf(fptr, "%[^,],[", num);
        char *line = (char*) malloc(sizeof(int));
```

```
        n = atoi(num);
        int *arr = (int*) calloc(n, sizeof(int));
        for (int i = 0; i < n-1; i++){
            fscanf(fptr, "%s", line);
            arr[i] = atoi(line);
        }
        fscanf(fptr, "%s]\n", line);
        arr[n-1] = atoi(line);
        printArray(arr, n);
        radix_exchange_sort(arr, n);
        printArray(arr, n);
        free(arr);
        free(line);
        free(num);
    }
    fclose(fptr);
    return 0;
}
```

INTERVAL / BUCKET SORT INT

```
typedef struct node* NODE;
struct node
{
    int ele;
    NODE next;
};
typedef struct linked_list* LIST;
struct linked_list
{
    int count;
    NODE head;
};
LIST createNewList(){
    LIST myList;
    myList = (LIST) malloc(sizeof(struct linked_list));
    if (myList == NULL){
        printf("Unable to allocate memory.\n");
        exit(1);
    }
    myList->count = 0;
    myList->head = NULL;
    return myList;
}
NODE createNewNode(int value){
    NODE myNode;
    myNode = (NODE) malloc(sizeof(struct node));
    if (myNode == NULL){
        printf("Unable to allocate memory.\n");
        exit(1);
    }
    myNode->ele = value;
    myNode->next = NULL;
    return myNode;
}
void printList(LIST l1)
```

```
{
    NODE temp = l1->head;
    printf("[HEAD] ->");
    while(temp!=NULL){
        printf(" %d ->", temp->ele);
        temp = temp->next;
    }
    printf("[NULL]\n");
}
void insertFirst(NODE value, LIST l1){
    value->next = l1->head;
    l1->head = value;
    l1->count++;
}
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;
        /* Move elements of arr[0..i-1],
          that are greater than key,
          to one position ahead of
          their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void sortList(LIST l1)
{
    int arr[l1->count];
    int i = 0;
    NODE temp = l1->head;
    while (temp != NULL)
    {
        arr[i] = temp->ele;
        i++;
        temp = temp->next;
    }
    insertionSort(arr, i);
    temp = l1->head;
    i = 0;
    while (temp != NULL)
    {
        temp->ele = arr[i];
        i++;
        temp = temp->next;
    }
}
void intervalSort(int arr[], int n, int m)
```

```
{
    int i, j;
    // Determine the number of buckets based on the maximum value in the array
    // int buckets = m > n ? n : m;
    // hard coding the value 10
    // int buckets = m/10;
    // possible scheme for number of buckets
    int buckets = sqrt(m);
    LIST b[buckets];
    for(i=0; i<buckets; i++)
    {
        b[i] = createNewList();
    }
    // Put array elements in different buckets
    for(i=0; i<n; i++)
    {
        // Calculate the bucket index based on the value and the number of buckets
        int bucketIndex = (arr[i] * buckets) / (m + 1);
        insertFirst(createNewNode(arr[i]), b[bucketIndex]);
    }
    // Sort individual buckets
    for(i=0; i<buckets; i++)
    {
        sortList(b[i]);
    }
    // Concatenate all buckets (in sequence) into arr[]
    for(i=0, j=0; i<buckets; i++)
    {
        NODE temp = b[i]->head;
        while(temp != NULL)
        {
            arr[j++] = temp->ele;
            temp = temp->next;
        }
    }
}
void printArray(int arr[], int n)
{
    printf("The array is : \n");
    for (int i = 0; i < n; i++)
    {
      printf(" %d", arr[i]);
    }
    printf("\n");
}
int main()
{
    srand(time(NULL));
    int n, m;
    printf("Enter the number of elements in array and range of numbers\n");
    scanf("%d %d", &n, &m);
    int *arr = (int*) malloc(sizeof(int)*n);
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand() % m;
```

```
    }
    printArray(arr, n);
    intervalSort(arr, n, m);
    printArray(arr, n);
    return 0;
}
```

INTERVAL / BUCKET SORT FLOAT

```
typedef struct node* NODE;
struct node
{
    float ele;
    NODE next;
};
typedef struct linked_list* LIST;
struct linked_list
{
    int count;
    NODE head;
};
LIST createNewList(){
    LIST myList;
    myList = (LIST) malloc(sizeof(struct linked_list));
    if (myList == NULL){
        printf("Unable to allocate memory.\n");
        exit(1);
    }
    myList->count = 0;
    myList->head = NULL;
    return myList;
}
NODE createNewNode(float value){
    NODE myNode;
    myNode = (NODE) malloc(sizeof(struct node));
    if (myNode == NULL){
        printf("Unable to allocate memory.\n");
        exit(1);
    }
    myNode->ele = value;
    myNode->next = NULL;
    return myNode;
}
void insertFirst(NODE value, LIST l1){
    value->next = l1->head;
    l1->head = value;
    l1->count++;
}
void sortList(LIST l1){
    NODE temp1 = l1->head;
    float temp;
    NODE temp2;
    while (temp1 != NULL) {
        temp2 = temp1->next;
        while (temp2 != NULL) {
        if (temp1->ele > temp2->ele) {
```

```
            temp = temp1->ele;
            temp1->ele = temp2->ele;
            temp2->ele = temp;
        }
        temp2 = temp2->next;
        }
        temp1 = temp1->next;
    }
}
void printArray(float arr[], int n)
{
    printf("The array is : \n");
    for (int i = 0; i < n; i++)
    {
      printf(" %lf", arr[i]);
    }
    printf("\n");
}
void intervalSort(float arr[], int n)
{
    int i, j;
    // Create n empty buckets
    LIST b[n];
    for(i=0; i<n; i++)
    {
        b[i] = createNewList();
    }
    // Put array elements in different buckets
    for(i=0; i<n; i++)
    {
        insertFirst(createNewNode(arr[i]), b[(int)(n*arr[i])]);
    }
    // Sort individual buckets
    for(i=0; i<n; i++)
    {
        sortList(b[i]); // sortList() function has to be implemented
    }
    // Concatenate all buckets (in sequence) into arr[]
    for(i=0, j=0; i<n; i++)
    {
        NODE temp = b[i]->head;
        while(temp != NULL)
        {
            arr[j++] = temp->ele;
            temp = temp->next;
        }
    }
}
int main()
{
    srand(time(NULL));
    int n;
    printf("Enter the number of elements in array and range of numbers\n");
    scanf("%d", &n);
    float *arr = (float*) malloc(sizeof(float)*n);
```

```
    for (int i = 0; i < n; i++)
    {
        arr[i] = (float) (rand() % 100) / 100;
    }
    printArray(arr, n);
    intervalSort(arr, n);
    printArray(arr, n);
    return 0;
}
```

LAB SHEET 7 TASK 5

```
char* intervalSort(char arr[], int n) {
    int count[128] = {0};
    for (int i = 0; i < n; i++) {
        count[(int)arr[i]]++;
    }
    for (int i = 1; i < 128; i++) {
        count[i] += count[i - 1];
    }
    char* sortedArray = (char*)malloc(n * sizeof(char));
    if (sortedArray == NULL) {
        printf("Memory allocation failed.\n");
        exit(1);
    }
    for (int i = n - 1; i >= 0; i--) {
        sortedArray[count[(int)arr[i]] - 1] = arr[i];
        count[(int)arr[i]]--;
    }
    return sortedArray;
}
int main() {
    int n;
    printf("Enter the length of String: ");
    scanf("%d", &n);
    char *A = (char *)malloc(n * sizeof(char));
    if (A == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    printf("Enter the String: ");
    scanf("%s", A);

    printf("The sorted array is: ");
    char* sortedA = intervalSort(A, n);
    printf("%s\n", sortedA);

    free(A);
    free(sortedA);

    return 0;
}
```

LAB SHEET 7 TASK 5 VERSION 2

```
void insertionSort(int arr[], int n)
{
    int i, key, j;
```

```
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1],
           that are greater than key,
           to one position ahead of
           their current position */
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
void sortList(LIST l1)
{
    int arr[l1->count];
    int i = 0;
    NODE temp = l1->head;
    while (temp != NULL)
    {
        arr[i] = temp->ele;
        i++;
        temp = temp->next;
    }
    insertionSort(arr, i);
    temp = l1->head;
    i = 0;
    while (temp != NULL)
    {
        temp->ele = arr[i];
        i++;
        temp = temp->next;
    }
}
void intervalSort(int arr[], int n, int noOfBuckets)
{
    int i, j;
    // Create n empty buckets
    LIST b[noOfBuckets];
    for (i = 0; i < noOfBuckets; i++)
    {
        b[i] = createNewList();
    }
    printf("new list created\n");
    // Put array elements in different buckets
    for (i = 0; i < n; i++)
    {
        insertFirst(b[(int)(arr[i]-65)], createNewNode(arr[i]));
    }
    printf("array elements inserted.\n");
    // Sort individual buckets
```

```
    for (i = 0; i < noOfBuckets; i++)
    {
        sortList(b[i]); // sortList() function has to be implemented
    }
    printf("individual buckets sorted.\n");
    // Concatenate all buckets (in sequence) into arr[]
    for (i = 0, j = 0; i <noOfBuckets; i++)
    {
        NODE temp = b[i]->head;
        while (temp != NULL)
        {
            arr[j++] = temp->ele;
            temp = temp->next;
        }
    }
    printf("line 225.\n");
    for(int i=0;i<n;i++){
        printf("%d ",arr[i]);
    }
}
int main(){
    char arr[50];
    printf("Enter a string of max 50 characters.\n");
    scanf("%s",arr);
    //printf("%s",arr);
    int n=strlen(arr);
    int arr2[n];
    for(int i=0;i<n;i++){
        arr2[i]=(int)arr[i];
        printf("%d ",arr2[i]);
    }
    int k = arr2[0];
    for (int i = 1; i < n; i++)
    {
        if (arr2[i] > k)
        {
            k = arr2[i];
        }
    }
    printf("\n%d",k);
    int B[n];
    intervalSort(arr2,n,58); //change
    printf("\n");
    for(int i=0;i<n;i++){
        arr[i]=(char)arr2[i];
        printf("%c",arr[i]);
    }
    //printf("%d",n);
}
```

LAB SHEET 7 TASK 5 VERSION 2

```
void intervalSort(float arr[], int n)
{
    int i, j;
    LIST b[n];
```

```
    for (i = 0; i < n; i++)
    {
        b[i] = createNewList();
        printf("Bucket %d created.\n", i);
    }
    float min_value = arr[0], max_value = arr[0];
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max_value)
        {
            max_value = arr[i];
        }
        if (arr[i] < min_value)
        {
            min_value = arr[i];
        }
    }
    for (i = 0; i < n; i++)
    {
        float scaled_value = (arr[i] - min_value) / (max_value - min_value);
        int index = (int)(scaled_value * (n - 1));
        insertFirst(b[index], createNewNode(arr[i]));
    }
    for (i = 0; i < n; i++)
    {
        sortList(b[i]);
    }
    for (i = 0, j = 0; i < n; i++)
    {
        NODE current = b[i]->head;
        while (current != NULL)
        {
            arr[j++] = current->d;
            current = current->next;
        }
    }
}
```

LINKED LIST SORT

```
void sortList(LIST l1){
    NODE temp1 = l1->head;
    int temp;
    NODE temp2;
    while (temp1 != NULL) {
        temp2 = temp1->next;
        while (temp2 != NULL) {
        if (temp1->ele > temp2->ele) {
            temp = temp1->ele;
            temp1->ele = temp2->ele;
            temp2->ele = temp;
        }
        temp2 = temp2->next;
        }
        temp1 = temp1->next;
    }
```

}
# LINKED LIST SORT WITH INSERTION SORT DONE ABOVE
# LINKED LIST SORT BUBBLE
```
#include <stdio.h>
#include <stdlib.h>

/* structure for a node */
struct Node {
    int data;
    struct Node* next;
} Node;

/*Function to swap the nodes */
struct Node* swap(struct Node* ptr1, struct Node* ptr2)
{
    struct Node* tmp = ptr2->next;
    ptr2->next = ptr1;
    ptr1->next = tmp;
    return ptr2;
}
/* Function to sort the list */
int bubbleSort(struct Node** head, int count)
{
    struct Node** h;
    int i, j, swapped;
    for (i = 0; i <= count; i++) {
        h = head;
        swapped = 0;
        for (j = 0; j < count - i - 1; j++) {
            struct Node* p1 = *h;
            struct Node* p2 = p1->next;
            if (p1->data > p2->data) {
                /* update the link after swapping */
                *h = swap(p1, p2);
                swapped = 1;
            }
            h = &(*h)->next;
        }
        /* break if the loop ended without any swap */
        if (swapped == 0)
            break;
    }
}
/* Function to print the list */
void printList(struct Node* n)
{
    while (n != NULL) {
        printf("%d -> ", n->data);
        n = n->next;
    }
    printf("\n");
}
/* Function to insert a struct Node
at the beginning of a linked list */
```

```
void insertAtTheBegin(struct Node** start_ref, int data)
{
    struct Node* ptr1
        = (struct Node*)malloc(sizeof(struct Node));

    ptr1->data = data;
    ptr1->next = *start_ref;
    *start_ref = ptr1;
}
```

LINKED LIST SORT INSERTION

```
struct node {
    int data;
    struct node* next;
};

struct node* head = NULL;
struct node* sorted = NULL;

void push(int val)
{
    /* allocate node */
    struct node* newnode
        = (struct node*)malloc(sizeof(struct node));
    newnode->data = val;
    /* link the old list of the new node */
    newnode->next = head;
    /* move the head to point to the new node */
    head = newnode;
}

/*
 * function to insert a new_node in a list. Note that
 * this function expects a pointer to head_ref as this
 * can modify the head of the input linked list
 * (similar to push())
 */
void sortedInsert(struct node* newnode)
{
    /* Special case for the head end */
    if (sorted == NULL || sorted->data >= newnode->data) {
        newnode->next = sorted;
        sorted = newnode;
    }
    else {
        struct node* current = sorted;
        /* Locate the node before the point of insertion
         */
        while (current->next != NULL
               && current->next->data < newnode->data) {
            current = current->next;
        }
        newnode->next = current->next;
        current->next = newnode;
    }
```

```
}
// function to sort a singly linked list
// using insertion sort
void insertionsort()
{
    struct node* current = head;

    // Traverse the given linked list and insert every
    // node to sorted
    while (current != NULL) {
        // Store next for next iteration
        struct node* next = current->next;
        // insert current in sorted linked list
        sortedInsert(current);
        // Update current
        current = next;
    }
    // Update head to point to sorted linked list
    head = sorted;
}
```

LINKED LIST SORT MERGE

```
/* Link list node */
struct Node {
    int data;
    struct Node* next;
};
/* function prototypes */
struct Node* SortedMerge(struct Node* a, struct Node* b);
void FrontBackSplit(struct Node* source,
                    struct Node** frontRef,
                    struct Node** backRef);
/* sorts the linked list by changing next pointers (not
 * data) */
void MergeSort(struct Node** headRef)
{
    struct Node* head = *headRef;
    struct Node* a;
    struct Node* b;
    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL)) {
        return;
    }
    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);
    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);
    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
/* See https:// www.geeksforgeeks.org/?p=3622 for details of
this function */
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
```

```
    struct Node* result = NULL;
    /* Base cases */
    if (a == NULL)
        return (b);
    else if (b == NULL)
        return (a);
    /* Pick either a or b, and recur */
    if (a->data <= b->data) {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return (result);
}
/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves, and return the two lists using
the reference parameters. If the length is odd, the extra node should go in the front list. Uses
the fast/slow pointer strategy. */
void FrontBackSplit(struct Node* source,
                    struct Node** frontRef,
                    struct Node** backRef)
{
    struct Node* fast;
    struct Node* slow;
    slow = source;
    fast = source->next;
    /* Advance 'fast' two nodes, and advance 'slow' one node
     */
    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }
    /* 'slow' is before the midpoint in the list, so split
    it in two at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}
/* Function to print nodes in a given linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}
/* Function to insert a node at the beginning of the linked
 * list */
void push(struct Node** head_ref, int new_data)
```

```
{
    /* allocate node */
    struct Node* new_node
        = (struct Node*)malloc(sizeof(struct Node));
    /* put in the data */
    new_node->data = new_data;
    /* link the old list of the new node */
    new_node->next = (*head_ref);
    /* move the head to point to the new node */
    (*head_ref) = new_node;
}
```

LINKED LIST SORT QUICK

```
// Creating structure
struct Node {
    int data;
    struct Node* next;
};
// Add new node at end of linked list
void insert(struct Node** head, int value)
{
    // Create dynamic node
    struct Node* node
        = (struct Node*)malloc(sizeof(struct Node));
    if (node == NULL) {
        // checking memory overflow
        printf("Memory overflow\n");
    }
    else {
        node->data = value;
        node->next = NULL;
        if (*head == NULL) {
            *head = node;
        }
        else {
            struct Node* temp = *head;
            // finding last node
            while (temp->next != NULL) {
                temp = temp->next;
            }
            // adding node at last position
            temp->next = node;
        }
    }
}
// Displaying linked list element
void display(struct Node* head)
{
    if (head == NULL) {
        printf("Empty linked list");
        return;
    }
    struct Node* temp = head;
    printf("\n Linked List :");
    while (temp != NULL) {
```

```
        printf(" %d", temp->data);
        temp = temp->next;
    }
}

// Finding last node of linked list
struct Node* last_node(struct Node* head)
{
    struct Node* temp = head;
    while (temp != NULL && temp->next != NULL) {
        temp = temp->next;
    }
    return temp;
}
// We are Setting the given last node position to its proper position
struct Node* partition(struct Node* first, struct Node* last)
{
    // Get first node of given linked list
    struct Node* pivot = first;
    struct Node* front = first;
    int temp = 0;
    while (front != NULL && front != last) {
        if (front->data < last->data) {
            pivot = first;
            // Swapping node values
            temp = first->data;
            first->data = front->data;
            front->data = temp;
            // Visiting the next node
            first = first->next;
        }
        // Visiting the next node
        front = front->next;
    }
    // Change last node value to current node
    temp = first->data;
    first->data = last->data;
    last->data = temp;
    return pivot;
}
// Performing quick sort in the given linked list
void quick_sort(struct Node* first, struct Node* last)
{
    if (first == last) {
        return;
    }
    struct Node* pivot = partition(first, last);
    if (pivot != NULL && pivot->next != NULL) {
        quick_sort(pivot->next, last);
    }

    if (pivot != NULL && first != pivot) {
        quick_sort(first, pivot);
    }
}
```

AVL

```
BST *new_bst()
{
    BST* bst = (BST*) malloc(sizeof(BST));
    bst->root = NULL;
    return bst;
}
Node *new_node(int value)
{
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->left = NULL;
    node->right = NULL;
    return node;
}
// task 1
Node *rotate_left(Node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}
Node *rotate_right(Node *y)
{
    struct node *x = y->left;
    y->left = x->right;
    x->right= y;
    return x;
}
int is_height_balanced(Node *x)
{
    int left = height(x->left);
    int right = height(x->right);
    if (left - right > 1 || right - left > 1) return -1;
    else return left>right ? left : right + 1;
}
Node *insertAVL(Node *node, int value)
{
    if (node == NULL)
    {
        node = new_node(value);
    }
    else if (value < node->value)
    {
        node->left = insertAVL(node->left, value);
    }
    else
    {
        node->right = insertAVL(node->right, value);
    }
    int balance = is_height_balanced(node);
    if (balance == -1)
    {
```

```
        if (value < node->value)
        {
            if (value < node->left->value)
            {
                // LL imbalance
                node = rotate_right(node);
            }
            else
            {
                // LR imbalance
                node->left = rotate_left(node->left);
                node = rotate_right(node);
            }
        }
        else
        {
            /*
            Complete the code for the following cases:
            RR imbalance
            RL imbalance
            */
            if (value < node->right->value)
            {
                //RL imbalance
                node->right = rotate_right(node->right);
                node = rotate_left(node);
            }
            else
            {
                //RR imbalance
                node = rotate_left(node);
            }
        }
    }
    return node;
}
void traverse_bfs(Node *node)
{
    if (node == NULL)
    {
        return;
    }
    Node *queue[100];
    int front = 0;
    int back = 0;
    queue[back++] = node;
    while (front != back)
    {
        Node *current = queue[front++];
        printf("%d ", current->value);
        if (current->left != NULL)
        {
            queue[back++] = current->left;
        }
        if (current->right != NULL)
```

```
        {
            queue[back++] = current->right;
        }
    }
}
int AVLCheck(Node *node)
{
    if (node == NULL)
        return 1;
    if (node->left != NULL && (maxValue(node->left) > node->value ||
is_height_balanced(node->left) == -1))
        return 0;
    if (node->right != NULL && (minValue(node->right) < node->value ||
is_height_balanced(node->right) == -1))
        return 0;
    if (!AVLCheck(node->left) || !AVLCheck(node->right))
        return 0;
    return 1;
}
int main()
{
    BST *avl = new_bst();
    avl->root = insertAVL(avl->root, 1);
    avl->root = insertAVL(avl->root, 2);
    avl->root = insertAVL(avl->root, 3);
    avl->root = insertAVL(avl->root, 4);
    avl->root = insertAVL(avl->root, 5);
    avl->root = insertAVL(avl->root, 6);
    avl->root = insertAVL(avl->root, 7);
    avl->root = insertAVL(avl->root, 8);
    avl->root = insertAVL(avl->root, 9);
    traverse_bfs(avl->root);
    printf("\n");
    AVLCheck(avl->root)?printf("It is an AVL tree!\n"):printf("It is not an AVL tree!\n");
    BST *fake_avl = new_bst();
    insert(fake_avl, 6);
    insert(fake_avl, 5);
    insert(fake_avl, 4);
    insert(fake_avl, 3);
    insert(fake_avl, 2);
    insert(fake_avl, 1);
    traverse_bfs(fake_avl->root);
    printf("\n");
    AVLCheck(fake_avl->root)?printf("It is an AVL tree!\n"):printf("It is not an AVL tree!\n");
    Return 0;
}
```

AVL WITH HEIGHT

```
typedef struct node
{
    int value;
    struct node *left;
    struct node *right;
    int height;
} Node;
```

```
typedef struct bst
{
    Node *root;
} BST;
BST *new_bst()
{
    BST *bst = malloc(sizeof(BST));
    bst->root = NULL;
    return bst;
}

struct node *new_node(int value)
{
    struct node *newNode = (struct node *)malloc(sizeof(struct node));
    newNode->value = value;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->height = 1; // Height initialized to 1 for leaf nodes
    return newNode;
}
void pre_iter(Node *root)
{
    if (root == NULL)
        return;
    Stack *s = createStack();
    push(s, root);

    while (!empty(s))
    {
        Node *current = pop(s);
        printf("Visited %d\n ", current->value);
        if (current->right != NULL)
            push(s, current->right);
        if (current->left != NULL)
            push(s, current->left);
    }
    freeStack(s);
}
void postiter(Node *root)
{
    if (root == NULL)
        return;
    struct Stack *stack = createStack();
    do
    {
        while (root)
        {
            if (root->right)
                push(stack, root->right);
            push(stack, root);
            root = root->left;
        }
        root = pop(stack);
        if (root->right && top(stack) == root->right)
        {
```

```
            pop(stack);
            push(stack, root);
            root = root->right;
        }
        else
        {
            printf("%d ", root->value);
            root = NULL;
        }
    } while (!empty(stack));
}
int height(struct node *n)
{
    if (n == NULL)
    {
        return 0;
    }
    return n->height;
}
int balance_factor(struct node *n)
{
    if (n == NULL)
    {
        return 0;
    }
    return height(n->left) - height(n->right);
}
struct node *rotate_right(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;
    x->right = y;
    y->left = T2;
    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}
struct node *rotate_left(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;
    y->left = x;
    x->right = T2;
    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}
struct node *insertAVL(struct node *node, int value)
{
    if (node == NULL)
    {
        return new_node(value);
    }
```

```
    if (value < node->value)
    {
        node->left = insertAVL(node->left, value);
    }
    else
    {
        node->right = insertAVL(node->right, value);
    }
    // Update height of current node
    node->height = 1 + max(height(node->left), height(node->right));

    int balance = balance_factor(node);
    // Perform rotations if tree becomes unbalanced
    if (balance > 1 && value < node->left->value)
    {
        return rotate_right(node);
    }
    if (balance < -1 && value >= node->right->value)
    {
        return rotate_left(node);
    }
    if (balance > 1 && value >= node->left->value)
    {
        node->left = rotate_left(node->left);
        return rotate_right(node);
    }
    if (balance < -1 && value < node->right->value)
    {
        node->right = rotate_right(node->right);
        return rotate_left(node);
    }
    return node;
}
Node *deleteAVL(Node *node, int value) from labsheet
// Driver program to test above functions
int main()
{
    // printf("Height of the tree is %d\n", height(bst->root));

    // BST *bst = new_bst();
    Node *r = bst->root;
    r = insertAVL(r, 50);
    r = insertAVL(r, 30);
    r = insertAVL(r, 70);
    r = insertAVL(r, 20);
    r = insertAVL(r, 40);
    r = insertAVL(r, 60);
    r = insertAVL(r, 80);
    traverse_bfs(r);
    printf("\n");
    r = deleteAVL(r, 40);
    printf("\n");
    traverse_bfs(r);

    return 0;
```

```
}
// void insertNonRecursive(struct Node **root, int key)
// {
//     struct Node *newNode = newNode(key);
//     if (*root == NULL)
//     {
//         *root = newNode;
//         return;
//     }
//     struct Node *curr = *root;
//     struct StackNode *stack = NULL;
//     while (curr != NULL)
//     {
//         push(&stack, curr);
//         if (key < curr->key)
//             curr = curr->left;
//         else
//             curr = curr->right;
//     }
//     struct Node *parent = pop(&stack);
//     if (key < parent->key)
//         parent->left = newNode;
//     else
//         parent->right = newNode;
//     while (stack != NULL)
//     {
//         curr = pop(&stack);
//         curr->height = 1 + max(height(curr->left), height(curr->right));
//         int balance = getBalance(curr);
//         if (balance > 1 && key < curr->left->key)
//             curr = rightRotate(curr);
//         if (balance < -1 && key > curr->right->key)
//             curr = leftRotate(curr);
//         if (balance > 1 && key > curr->left->key)
//         {
//             curr->left = leftRotate(curr->left);
//             curr = rightRotate(curr);
//         }
//         if (balance < -1 && key < curr->right->key)
//         {
//             curr->right = rightRotate(curr->right);
//             curr = leftRotate(curr);
//         }
//         if (curr->left != NULL)
//             curr->left->height =
//                 1 + max(height(curr->left->left), height(curr->left->right));
//         if (curr->right != NULL)
//             curr->right->height =
//                 1 + max(height(curr->right->left), height(curr->right->right));
//         if (stack != NULL)
//         {
//             parent = stack->node;
//             if (curr->key < parent->key)
//                 parent->left = curr;
//             else
```

```
//                parent->right = curr;
//         }
//         else
//         {
//             *root = curr;
//         }
//     }
// }
// void deleteNonRecursive(struct Node **root, int key)
// {
//     struct Node *curr = *root;
//     struct Node *parent = NULL;
//     struct StackNode *stack = NULL;
//     // Search for the node to delete
//     while (curr != NULL && curr->key != key)
//     {
//         push(&stack, curr);
//         parent = curr;
//         if (key < curr->key)
//             curr = curr->left;
//         else
//             curr = curr->right;
//     }

//     if (curr == NULL)
//         return; // Key not found

//     // Case 1: Node to delete has no children
//     if (curr->left == NULL && curr->right == NULL)
//     {
//         if (parent == NULL)
//         {
//             *root = NULL;
//             free(curr);
//             return;
//         }
//         if (parent->left == curr)
//             parent->left = NULL;
//         else
//             parent->right = NULL;
//         free(curr);
//     }
//     // Case 2: Node to delete has one child
//     else if (curr->left == NULL || curr->right == NULL)
//     {
//         struct Node *child = (curr->left != NULL) ? curr->left : curr->right;
//         if (parent == NULL)
//         {
//             *root = child;
//             free(curr);
//             return;
//         }
//         if (parent->left == curr)
//             parent->left = child;
//         else
```

```
//            parent->right = child;
//        free(curr);
//    }
//    // Case 3: Node to delete has two children
//    else
//    {
//        struct Node *succParent = curr;
//        struct Node *succ = curr->right;
//        while (succ->left != NULL)
//        {
//            push(&stack, succ);
//            succParent = succ;
//            succ = succ->left;
//        }
//        curr->key = succ->key;
//        if (succParent == curr)
//            succParent->right = succ->right;
//        else
//            succParent->left = succ->right;
//        free(succ);
//    }
//    while (stack != NULL)
//    {
//        curr = pop(&stack);
//        curr->height = 1 + max(height(curr->left), height(curr->right));
//        int balance = getBalance(curr);
//        if (balance > 1 && getBalance(curr->left) >= 0)
//            curr = rightRotate(curr);
//        if (balance > 1 && getBalance(curr->left) < 0)
//        {
//            curr->left = leftRotate(curr->left);
//            curr = rightRotate(curr);
//        }
//        if (balance < -1 && getBalance(curr->right) <= 0)
//            curr = leftRotate(curr);
//        if (balance < -1 && getBalance(curr->right) > 0)
//        {
//            curr->right = rightRotate(curr->right);
//            curr = leftRotate(curr);
//        }
//        if (curr->left != NULL)
//            curr->left->height =
//                1 + max(height(curr->left->left), height(curr->left->right));
//        if (curr->right != NULL)
//            curr->right->height =
//                1 + max(height(curr->right->left), height(curr->right->right));
//        if (stack != NULL)
//        {
//            parent = stack->node;
//            if (curr->key < parent->key)
//                parent->left = curr;
//            else
//                parent->right = curr;
//        }
//        else
```

```
//         {
//             *root = curr;
//         }
//     }
// }
// struct StackNode
// {
//     struct Node *node;
//     struct StackNode *next;
// };
// void push(struct StackNode **root, struct Node *node)
// {
//     struct StackNode *stackNode = createStackNode(node);
//     stackNode->next = *root;
//     *root = stackNode;
// }
// struct Node *pop(struct StackNode **root)
// {
//     if (*root == NULL)
//         return NULL;
//     struct StackNode *temp = *root;
//     *root = (*root)->next;
//     struct Node *popped = temp->node;
//     free(temp);
//     return popped;
// }
```

AVL WITH PARENT

```
 #define max(a,b) \
   ({ __typeof__ (a) _a = (a); \
       __typeof__ (b) _b = (b); \
     _a > _b ? _a : _b; })
typedef struct node {
   char *str;
   int nb;
   struct node * parent, * left, * right;
} node_t;
#define DEBUG false
void tree_render_node( node_t * node ) {
        if ( ! node )
                return;
        tree_render_node ( node->left );
        printf("* %20s   / ", node->str );
        if ( node->parent ) {
                printf("parent = %s, ", node->parent->str );
        }
        if ( node->left ) {
                printf("left = %s, ", node->left->str );
        }
        if ( node->right ) {
                printf("right = %s, ", node->right->str );
        }
        printf("\n");
        tree_render_node ( node->right );
}
```

```
void tree_render( node_t * tree ) {
	printf("Tree:\n");
	tree_render_node( tree );
	printf("\n");
}
int tree_depth( node_t * node ) {
	if ( node )
		return 1 + max( tree_depth( node->left ), tree_depth( node->right ) );
	else
		return 0;
}
int tree_balance_factor( node_t * node ) {
	int left = tree_depth( node->left );
	int right = tree_depth( node->right );
	int balance = left - right;
	//printf("%p (%s) / left = %d (%s), right = %d (%s) / balance = %d\n", node, node->str,
left, node->left ? node->left->str : NULL, right, node->right ? node->right->str : NULL,
balance );
	return balance;
}
void tree_rotate_left( node_t ** node ) {
	node_t * top = *node;
	node_t * right = top->right;
	node_t * x = right->left;
	if ( DEBUG ) printf("Rotating left on %s...\n", (*node)->str);
	*node = right;
	top->right = right->left;
	right->left = top;
	if ( x )
		x->parent = top;
	right->parent = top->parent;
	top->parent = right;
}
void tree_rotate_right( node_t ** node ) {
	node_t * top = *node;
	node_t * left = top->left;
	node_t * x = left->right;
	if ( DEBUG ) printf("Rotating right on %s...\n", (*node)->str);
	*node = left;
	top->left = left->right;
	left->right = top;
	if ( x )
		x->parent = top;
	left->parent = top->parent;
	top->parent = left;
}
void tree_balance( node_t ** node ) {
	int bf = tree_balance_factor( *node );
	if ( bf == 2 ) {
		if( tree_balance_factor( (*node)->left ) == -1 ) {
			tree_rotate_left( & (*node)->left );
		}
		tree_rotate_right( node );
	}
	else if ( bf == -2 ) {
```

```
			if( tree_balance_factor( (*node)->right ) == 1 ) {
				tree_rotate_right( & (*node)->right );
			}
			tree_rotate_left( node );
		}
}
bool tree_insert( node_t ** target, node_t * node ) {
		node->left = NULL;
		node->right = NULL;
		node->parent = NULL;
		node_t ** ancestor = target;

		if ( ! * target ) {
			*target = node;
			return true;
		}
		else {
			target = strcmp( (*target)->str, node->str ) < 0 ? & (*target)->right : & (*target)-
>left;
			if ( tree_insert( target, node ) ) {
				node->parent = *ancestor;
			}
			tree_balance( ancestor );
			return false;
		}
}
node_t * tree_search( node_t * branch, char * str, int * depth ) {
		*depth = 0;
		while( branch ) {
			int c = strcmp( branch->str, str );
			if ( c < 0 ) {
				branch = branch->right;
			} else if ( c > 0 ) {
				branch = branch->left;
			} else {
				break;
			}
			*depth += 1;
		}
		return branch;
}
node_t * get_next_node( node_t * node ) {
		if ( node->right ) {
			node = node->right;
			while( node->left ) {
				node = node->left;
			}
			return node;
		} else {
			node_t * parent = node->parent;
			while( parent ) {
				if ( strcmp(parent->str, node->str) > 0 )
					return parent;
				parent = parent->parent;
			}
```

```
			return parent;
		}
}
void tree_search_render( node_t * branch, char * str ) {
		int depth;
		node_t * node = tree_search( branch , str, & depth );
		printf("Searching \"%s\" : ", str );
		if ( node ) {
			printf(" str=\"%s\", nb = %d", node->str, node->nb );
		}
		else {
			printf("NOT FOUND");
		}
		printf(", depth = %d", depth );
		printf("\n");
}
int tree_max_depth( node_t * node ) {
		if ( ! node )
			return 0;
		return max( tree_max_depth( node->right ), tree_max_depth( node->left ) ) + 1;
}
int main() {
		char * names [] = { "Florent",  "Louis", "Marie", "Charles", "Mathilde", "Lucie",
"Fabienne", "Jean", "Patrick", "Nicolas", "Bernard", "Jacqueline", "Helene", "Guillaume",
"Anam", "Jules", "Jeanne", "Elton", "Michael", "Jean-Paul", "Matthieu", "Trung", "Zhen",
"Aline", "Anouk", "Adrien", "Alfred", "Sylvain", "Basile", "Constance", "A", "B", "C", "D", "E",
"F", "G", "H", "I", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z" };
		int nb = sizeof(names)/sizeof(char *);
		node_t nodes[nb];
		node_t * tree = NULL;
		{
			int i;
			for( i = 0; i < nb; i++ ) {
				node_t * node = & nodes[i];
				node->str = names[i];
			node->nb = i;
				if ( DEBUG ) printf("Inserting \"%s\" : %p\n", node->str, node );
				tree_insert( & tree, node );
			}
		}
		printf("Depth: %d\n", tree_max_depth( tree ) );
		tree_render( tree );
		{
			tree_search_render(tree, "Jerome");
			tree_search_render(tree, "Charles");
		}
		{
			int depth;
			node_t * n = tree_search( tree, "Bernard", & depth );
			while( n ) {
				printf("* %s\n", n->str);
				n = get_next_node( n );
			}
		}
```

```
        return 0;
}
```

HEAP

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
// compile with -lm flag
struct heap {
    int *data;
    int size;
    int capacity;
    int depth;
};
typedef struct heap* Heap;
Heap heap_create()
{
    Heap h = (Heap) malloc(sizeof(struct heap));
    h->data = (int *) malloc(sizeof(int));
    h->size = 0;
    h->capacity = 1;
    h->depth = 0;
    return h;
}
void add_to_tree(Heap h, int element)
{
    if (h->size >= h->capacity)
    {
        h->depth++;
        h->capacity = pow(2, h->depth+1) - 1;
        h->data = (int *) realloc(h->data, (h->capacity)*sizeof(int));
        h->data[h->size++] = element;
    }
    else
    {
        h->data[h->size++] = element;
    }
}
typedef struct node {
    int value;
    struct node *left;
    struct node *right;
} Node;
typedef struct bt {
    Node *root;
} BT;
BT *new_bst()
{
    BT *bt = malloc(sizeof(BT));
    bt->root = NULL;
    return bt;
}
Node *new_node(int value)
{
```

```
    Node *node = malloc(sizeof(Node));
    node->value = value;
    node->left = NULL;
    node->right = NULL;
    return node;
}
int parent(Heap h, int node)
{
    if (h->size < 2) return -1;
    else return (node-1)/2;
    // return (node - 1)/2;
}
int left_child(Heap h, int node)
{
    if (2*node + 1 < h->size) return 2*node + 1;
    else return -1;
    // return 2*node + 1;
}

int right_child(Heap h, int node)
{
    if (2*node + 2 < h->size) return 2*node + 2;
    else return -1;
    // return 2*node + 2;
}

void max_heapify(Heap h, int index)
{
    int left = left_child(h, index);
    int right = right_child(h, index);
    int largest = index;

    if (left < h->size && h->data[left] > h->data[largest])
    {
        largest = left;
    }
    if (right < h->size && h->data[right] > h->data[largest])
    {
        largest = right;
    }
    if (largest != index)
    {
        int temp = h->data[index];
        h->data[index] = h->data[largest];
        h->data[largest] = temp;
        max_heapify(h, largest);
    }
}

// task 3
void build_max_heap(Heap h)
{
    // for (int i = h->size - 1; i > -1; i--)
    // dont need to heapify leaves
    for (int i = h->capacity/2 -1; i > -1; i--)
```

```
    {
        // printf("Iteration %d \n", i);
        max_heapify(h, i);
    }
}

// task 4
int nodes_at_depth(Heap h, int depth)
{
    if (depth < h->depth) return pow(2, depth);
    else if (depth == h->depth) return h->size - pow(2, h->depth) + 1;
    else return 0;
}

void heap_sort(Heap h)
{
    build_max_heap(h);
    int temp = h->size;
    for (int i = h->size - 1; i >= 1; i--)
    {
        int temp = h->data[0];
        h->data[0] = h->data[i];
        h->data[i] = temp;
        h->size = h->size - 1;
        max_heapify(h, 0);
    }
    h->size = temp;
}

int main()
{
    Heap h1 = heap_create();
    add_to_tree(h1, 5);
    add_to_tree(h1, 7);
    add_to_tree(h1, 3);
    add_to_tree(h1, 2);
    add_to_tree(h1, 1);
    add_to_tree(h1, 10);
    add_to_tree(h1, 8);
    add_to_tree(h1, 0);
    add_to_tree(h1, 11);
    add_to_tree(h1, 14);

    for (int i = 0; i < h1->size; i++)
    {
        printf(" %d ", h1->data[i]);
    }

    printf("\n");
    printf("Depth = %d\n", h1->depth);
    printf("Capacity = %d\n", h1->capacity);
    printf("Size = %d\n", h1->size);

    // for (int i = 0; i < h1->size; i++)
    // {
```

```
    //     printf("\nValue = %d\n", h1->data[i]);
    //     printf("\nParent = %d\n", h1->data[parent(h1, i)]);
    //     printf("\nLeft child = %d\n", h1->data[left_child(h1, i)]);
    //     printf("\nRight child = %d\n", h1->data[right_child(h1, i)]);
    //     printf("\n");
    // }

    build_max_heap(h1);
    for (int i = 0; i < h1->size; i++)
    {
        printf(" %d ", h1->data[i]);
    }
    printf("\n");

    printf("Depth = %d\n", h1->depth);
    printf("Capacity = %d\n", h1->capacity);
    printf("Size = %d\n", h1->size);

    printf("Number of nodes at depth 2 = %d \n", nodes_at_depth(h1, 2));
    printf("Number of nodes at depth 3 = %d \n", nodes_at_depth(h1, 3));
    printf("Number of nodes at depth 4 = %d \n", nodes_at_depth(h1, 4));

    heap_sort(h1);

    for (int i = 0; i < h1->size; i++)
    {
        printf(" %d ", h1->data[i]);
    }
    printf("\n");

    return 0;
}
void min_heapify(Heap h, int index)
{
    int left = left_child(h, index);
    int right = right_child(h, index);
    int smallest = index;
    if (left < h->size && h->data[left] < h->data[smallest])
    {
        smallest = left;
    }
    if (right < h->size && h->data[right] < h->data[smallest])
    {
        smallest = right;
    }
    if (smallest != index)
    {
        int temp = h->data[index];
        h->data[index] = h->data[smallest];
        h->data[smallest] = temp;
        min_heapify(h, smallest);
    }
}

void build_min_heap(Heap h)
```

```
{
   for (int i = h->size - 1; i > -1; i--)
   // dont need to heapify leaves
   // for (int i = h->capacity/2 -1; i > -1; i--)
   {
      printf("Iteration %d \n", i);
      min_heapify(h, i);
   }
}
// extractMin definition
int extractMin(Heap h)
{
         int deleteItem;
         // Checking if the heap is empty or not
         if (h->size == 0) {
                  printf("\nHeap id empty.");
                  return -1;
         }
         // Store the node in deleteItem that
         // is to be deleted.
         deleteItem = h->data[0];
         // Replace the deleted node with the last node
         h->data[0] = h->data[h->size - 1];
         // Decrement the size of heap
         h->size--;
         // Call minheapify_top_down for 0th index
         // to maintain the heap property
         min_heapify(h, 0);
         return deleteItem;
}
```

HEAP WITH BINARY TREE

```
typedef struct node* Node;
struct node{
   int value;
   Node left;
   Node right;
};
typedef struct bt* BT;
struct bt
{
   int size;
   Node root;
};
int memused;
void* myalloc(int size)
{
   memused+=size;
   return malloc(size);
}
void* myrealloc(int old , void* ptr, int size)
{
   memused-=old;
   memused+=size;
   return realloc(ptr, size);
```

```
}
BT create_BT()
{
   BT new_BT = (BT)myalloc(sizeof(struct bt));
   new_BT->root = NULL;
   new_BT->size = 0;
   return new_BT;
}
Node create_Node(int value)
{
   Node new_node = (Node)myalloc(sizeof(struct node));
   new_node->left=NULL;
   new_node->right=NULL;
   new_node->value=value;
   return new_node;
}
void add_to_tree(BT bt, int value)
{
   Node new_node = create_Node(value);
   if(bt->size == 0)
   {
      bt->root= new_node;
      bt->size++;
      return;
   }
   int curr_size = bt->size+1;
   int num_bits=0;
   while (curr_size>0)
   {
      num_bits++;
      curr_size/=2;
   }
   curr_size = bt->size+1;
   Node curr = bt->root;
   for (int i = num_bits-2; i >=1; i--)
   {
      int temp = (curr_size>>i)&1;
      if(temp == 1)
      {
         curr=curr->right;
      }
      else
      {
         curr=curr->left;
      }
   }
   int lorr = curr_size&1;
   // printf("val %d \n", curr->value);
   if(lorr==0)
   {
      curr->left = new_node;
   }
   else
   {
      curr->right = new_node;
```

```
    }
    bt->size++;
}
void print_queue(Node* node, int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d ", node[i]->value);
    }
    printf("\n");

}
void print(BT bt)
{
    printf("Size: %d Space: %d\n" , bt->size, memused);
    if(bt->size==0)
    {
        return;
    }
    Node queue[bt->size];
    int l=0;
    int r=0;
    queue[r] = bt->root;
    r++;
    while(l!=r)
    {
        // print_queue(queue, bt->size);
        Node curr = queue[l];
        l++;
        printf("%d ", curr->value);
        if(curr->right!=NULL)
        {
            queue[r]=curr->left;
            r++;
            queue[r]=curr->right;
            r++;
        }
        else if (curr->left!=NULL)
        {
            queue[r]=curr->left;
            r++;
        }
    }
    printf("\n");
}
int main()
{
    BT h = create_BT();
    print(h);
    add_to_tree(h,0);
    print(h);
    add_to_tree(h,1);
    print(h);
    add_to_tree(h,2);
    print(h);
```

```
    add_to_tree(h,3);
    print(h);
    add_to_tree(h,4);
    print(h);
    add_to_tree(h,5);
    print(h);
}
```

MIN HEAP

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
struct heap
{
    int *data;
    int size;
    int capacity;
    int depth;
};
typedef struct heap *Heap;
Heap heap_create()
{
    Heap h = malloc(sizeof(struct heap));
    h->data = malloc(sizeof(int));
    h->size = 0;
    h->capacity = 1;
    h->depth = 0;
    return h;
}
void add_to_heap(Heap h, int value)
{
    if (h->size >= h->capacity)
    {
        h->depth++;
        h->capacity = pow(2, h->depth + 1) - 1;
        h->data = (int *)realloc(h->data, (h->capacity) * sizeof(int));
        h->data[h->size++] = value;
    }
    else
    {
        h->data[h->size++] = value;
    }
}
int parent(Heap h, int node)
{
    return (node - 1) / 2;
}
int left_child(Heap h, int node)
{
    return (2 * node + 1);
}
int right_child(Heap h, int node)
{
    return (2 * node + 2);
}
```

```
void min_heapify(Heap h, int index)
{
    int left = left_child(h, index);
    int right = right_child(h, index);
    int smallest = index;

    if (left < h->size && h->data[left] < h->data[smallest])
    {
        smallest = left;
    }
    if (right < h->size && h->data[right] < h->data[smallest])
    {
        smallest = right;
    }
    if (smallest != index)
    {
        int temp = h->data[index];
        h->data[index] = h->data[smallest];
        h->data[smallest] = temp;
        min_heapify(h, smallest);
    }
}
int minimum(Heap h)
{
    if (h->size == 0)
        return -1; // -1 denotes that the heap is empty
    else
    {
        return h->data[0];
    }
}
int extract_minimum(Heap h)
{
    int min = minimum(h);
    h->data[0] = h->data[h->size - 1];
    h->size--;
    min_heapify(h, 0);
    return min;
}
void decrease_key(Heap h, int x, int k)
{
    h->data[x] = k;
    int i = x;
    while (i != 0 && h->data[parent(h, i)] > h->data[i])
    {
        int temp = h->data[i];
        h->data[i] = h->data[parent(h, i)];
        h->data[parent(h, i)] = temp;
        i = parent(h, i);
    }
}
void printHeap(Heap h)
{
    for (int i = 0; i < h->size; i++)
    {
```

```
        printf("%d ", h->data[i]);
    }
    printf("\n");
}
Heap build_min_heap(int *a, int n)
{
    Heap h = heap_create();
    for (int i = 0; i < n; i++)
    {
        add_to_heap(h, a[i]);
    }
    for (int i = (h->capacity / 2) - 1; i >= 0; i--)
    {
        min_heapify(h, i);
    }
    return h;
}
int main()
{
    int a[10] = {11, 42, 53, 23, 44, 144, 76, 87, 90, 964};
    Heap h = build_min_heap(a, 10);
    printHeap(h);
    decrease_key(h, 3, 10);
    printHeap(h);
    printf("Minimum elements in ascending order: ");
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", extract_minimum(h));
    }
    printf("\n");
    return 0;
}
```

2-4 TREE

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node {
  int keys[3];
  struct node *children[4];
  int num_keys;
  int isLeaf;
} Node;
typedef struct tree {
  Node *root;
} Tree;
Node *new_node() {
  Node *temp = malloc(sizeof(Node));
  for (int i = 0; i < 3; i++) {
    temp->keys[i] = 0;
    temp->children[i] = NULL;
  }
  temp->children[3] = NULL;
  temp->isLeaf = 0;
  temp->num_keys = 0;
  return temp;
```

```
}
void insert_24(Tree *tree, int val) {
  Node *temp = tree->root;
  if (temp == NULL) {
    Node *myNode = new_node();
    myNode->isLeaf = 1;
    myNode->keys[0] = val;
    myNode->num_keys = 1;
    tree->root = myNode;
    return;
  }
  // traverse to leaf, splitting 4-nodes as we go
  // A 4-node is a node with 3 keys and 4 children
  Node *parent = NULL;
  while (temp) {
    if (temp->num_keys == 3) {
      // split 4-node
      Node *newNode = new_node();
      newNode->isLeaf = temp->isLeaf;
      newNode->children[0] = temp->children[2];
      newNode->children[1] = temp->children[3];
      newNode->keys[0] = temp->keys[2];
      newNode->num_keys = 1;
      temp->children[2] = NULL;
      temp->children[3] = NULL;
      temp->num_keys = 1;
      // insert new node into parent
      if (parent == NULL) {
        parent = new_node();
        parent->isLeaf = 0;
        parent->children[0] = temp;
        parent->children[1] = newNode;
        parent->keys[0] = temp->keys[1];
        parent->num_keys = 1;
        tree->root = parent;
        printf("Created new root node\n");
      } else
      // The parent must have 1 or 2 keys since all 3 nodes have been split
      {
        if (parent->num_keys == 1) {
          if (parent->keys[0] > temp->keys[1]) {
            parent->children[2] = parent->children[1];
            parent->children[1] = newNode;
            parent->keys[1] = parent->keys[0];
            parent->keys[0] = temp->keys[1];
          } else {
            parent->children[2] = newNode;
            parent->keys[1] = temp->keys[1];
          }
          parent->num_keys = 2;
        }
        // ...
        // 2 keys in parent
        else
        // ...
```

```
    {
      if (parent->keys[0] > temp->keys[1]) {
        parent->children[3] = parent->children[2];
        parent->children[2] = parent->children[1];
        parent->children[1] = newNode;
        parent->keys[2] = parent->keys[1];
        parent->keys[1] = parent->keys[0];
        parent->keys[0] = temp->keys[1];
      } else if (parent->keys[1] > temp->keys[1]) {
        parent->children[3] = parent->children[2];
        parent->children[2] = newNode;
        parent->keys[2] = parent->keys[1];
        parent->keys[1] = temp->keys[1];
      } else {
        parent->children[3] = newNode;
        parent->keys[2] = temp->keys[1];
      }
      parent->num_keys = 3;
    }
  }
  // Find the correct parent and child for the next iteration
  for (int i = 0; i <= parent->num_keys; i++) {
    if (val < parent->keys[i]) {
      parent = parent->children[i];
      break;
    } else if (i == parent->num_keys) {
      parent = parent->children[i];
      break;
    }
  }
  // Find the correct child for the next iteration
  for (int i = 0; i <= parent->num_keys; i++) {
    if (val < parent->keys[i]) {
      temp = parent->children[i];
      break;
    } else if (i == parent->num_keys) {
      temp = parent->children[i];
      break;
    }
  }
}
// If the node is not a 4-node, just traverse to the correct child
else {
  parent = temp;
  // Find the correct child for the next iteration
  for (int i = 0; i < parent->num_keys; i++) {
    if (val < parent->keys[i]) {
      temp = parent->children[i];
      break;
    }
  }
  if (parent == temp) {
    temp = parent->children[parent->num_keys];
  }
}
```

```
  }
  // Insert the value into the leaf
  for (int i = parent->num_keys - 1; i >= 0; i--) {
    if (val < parent->keys[i]) {
      parent->keys[i + 1] = parent->keys[i];
    } else {
      parent->keys[i + 1] = val;
      break;
    }
    if (i == 0) {
      parent->keys[i] = val;
    }
  }
  parent->num_keys++;
}
void print_tree(Node *node, int level) {
  printf("%*s", level * 2, "");
  if (node == NULL) {
    return;
  }
  if (node->isLeaf == 1) {
    for (int i = 0; i < node->num_keys; i++) {
      printf("%d ", node->keys[i]);
    }
    printf("\n");
  } else {
    for (int i = 0; i < node->num_keys; i++) {
      printf("%d ", node->keys[i]);
    }
    printf("\n");
    for (int i = 0; i <= node->num_keys; i++) {
      print_tree(node->children[i], level + 1);
    }
  }
}
Node *search(int key, Tree *tree) {
  Node *parent, *temp = tree->root;
  while (temp) {
    parent = temp;
    // Find the correct child for the next iteration
    for (int i = 0; i < parent->num_keys; i++) {
      if (key == parent->keys[i]) {
        return parent;
      } else if (key < parent->keys[i]) {
        temp = parent->children[i];
        break;
      }
    }
    if (parent == temp) {
      temp = parent->children[parent->num_keys];
    }
  }
  return temp;
}
```

```
int main() {
  Tree *tree = malloc(sizeof(Tree));
  tree->root = NULL;
  int keys[] = {4, 6, 14, 20, 2, 5, 10, 8, 15, 12, 18, 25, 16};
  for (int i = 0; i <= 12; i++) {
    insert_24(tree, keys[i]);
    printf("Tree after inserting %d\n", keys[i]);
    print_tree(tree->root, 0);
  }
  Node *sres = search(10, tree);
  if (sres)
    printf("Key found\n");
  else
    printf("Not found");
  sres = search(13, tree);
  if (sres)
    printf("Key found\n");
  else
    printf("Not found\n");
}
```