Max Heap

```
#include <stdio.h>
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
void maxHeapify(int arr[], int n, int i) {
   int largest = i;
   int left = 2 * i + 1;
   int right = 2 * i + 2;
   if (left < n && arr[left] > arr[largest])
      largest = left;
   if (right < n && arr[right] > arr[largest])
      largest = right;
   if (largest != i) {
      swap(&arr[i], &arr[largest]);
      maxHeapify(arr, n, largest);
   }
}
void buildMaxHeap(int arr[], int n) {
   int startIdx = (n / 2) - 1;
   for (int i = startIdx; i >= 0; i--) {
      maxHeapify(arr, n, i);
   }
}
void printArray(int arr[], int n) {
   for (int i = 0; i < n; ++i)
      printf("%d ", arr[i]);
   printf("\n");
```

```
}
int main() {
    int arr[] = { 10, 20, 15, 17, 9, 21 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: \n");
    printArray(arr, n);
    buildMaxHeap(arr, n);
    printf("Max Heap: \n");
    printArray(arr, n);
    return 0;
}
```

Min Heap

```
#include <stdio.h>
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void minHeapify(int arr[], int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] < arr[smallest])
        smallest = left;
    if (right < n && arr[right] < arr[smallest])
        smallest = right;
    if (smallest != i) {
        swap(&arr[i], &arr[smallest]);
        minHeapify(arr, n, smallest);
    }
```

```
}
void buildMinHeap(int arr[], int n) {
   int startIdx = (n / 2) - 1;
   for (int i = startIdx; i >= 0; i--) {
      minHeapify(arr, n, i);
   }
}
void printArray(int arr[], int n) {
   for (int i = 0; i < n; ++i)
      printf("%d ", arr[i]);
   printf("\n");
}
int main() {
   int arr[] = { 10, 20, 15, 17, 9, 21 };
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("Original array: \n");
   printArray(arr, n);
   buildMinHeap(arr, n);
   printf("Min Heap: \n");
   printArray(arr, n);
   return 0;
}
```

Connect n ropes with minimal cost

```
#include <stdio.h>
#include <stdlib.h>
int compare(const void* a, const void* b) {
   return (*(int*)a - *(int*)b);
}
int minCost(int ropes[], int n) {
   qsort(ropes, n, sizeof(int), compare);
   int totalCost = 0;
   while (n > 1) {
      int min1 = ropes[0];
      qsort(ropes + 1, n - 1, sizeof(int), compare);
      int min2 = ropes[1];
      int sum = min1 + min2;
      totalCost += sum;
      ropes[0] = sum;
      for (int i = 1; i < n - 1; i++) {
         ropes[i] = ropes[i + 1];
      }
      n--;
   }
   return totalCost;
}
int main() {
   int ropes[] = {5, 4, 2, 8};
   int n = sizeof(ropes) / sizeof(ropes[0]);
   printf("The minimum cost is %d\n", minCost(ropes, n));
   return 0;
}
```

Replace each array element by its corresponding rank

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct {
   int element;
   int index;
} Pair;
int compare(const void* a, const void* b) {
   return ((Pair*)a)->element - ((Pair*)b)->element;
}
void replaceWithRanks(int arr[], int n) {
   Pair pairs[n];
   for (int i = 0; i < n; i++) {
      pairs[i].element = arr[i];
      pairs[i].index = i;
   }
   qsort(pairs, n, sizeof(Pair), compare);
   int rank = 1;
   for (int i = 0; i < n; i++) {
      arr[pairs[i].index] = rank++;
   }
}
void printArray(int arr[], int n) {
   printf("[ ");
   for (int i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("]\n");
}
```

```
int main() {
    int arr[] = {10, 8, 15, 12, 6, 20, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original array: ");
    printArray(arr, n);
    replaceWithRanks(arr, n);
    printf("Array after replacing with ranks: ");
    printArray(arr, n);
    return 0;
}
```

Convert max heap to min heap in linear time

```
#include <stdio.h>
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void minHeapify(int arr[], int n, int i) {
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] < arr[smallest])
        smallest = left;
    if (right < n && arr[right] < arr[smallest])
        smallest = right;
    if (smallest != i) {
        swap(&arr[i], &arr[smallest]);
        minHeapify(arr, n, smallest);
    }
}
```

```
void convertMaxHeapToMinHeap(int arr[], int n) {
    for (int i = (n / 2) - 1; i >= 0; i--) {
        minHeapify(arr, n, i);
    }
}
void printArray(int arr[], int n) {
    printf("[ ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("]\n");
}
int main() {
    int arr[] = {9, 4, 7, 1, -2, 6, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("Original max-heap array: ");
    printArray(arr, n);
    convertMaxHeapToMinHeap(arr, n);
    printf("Min-heap array after conversion: ");
    printArray(arr, n);
    return 0;
}
```

Hashmaps

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#define TABLE_SIZE 10
typedef struct Node {
    char* key;
```

```
    int value;
    struct Node* next;
} Node;
Node* createNode(char* key, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = strdup(key);  // Duplicate the key string
    newNode->value = value;
    newNode->next = NULL;
    return newNode;
}
unsigned int hashFunction(char* key) {
    unsigned long int hashValue = 0;
    int i = 0;
    while (hashValue < ULONG_MAX && i < strlen(key)) {
        hashValue = hashValue << 8;
        hashValue += key[i];
        i++;
    }
    return hashValue % TABLE_SIZE;
}
void insert(Node* table[], char* key, int value) {
    unsigned int index = hashFunction(key);
    Node* newNode = createNode(key, value);
    if (table[index] == NULL) {
        table[index] = newNode;
    } else {
        Node* temp = table[index];
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
```

```
    }
}
int search(Node* table[], char* key) {
    unsigned int index = hashFunction(key);
    Node* temp = table[index];
    while (temp != NULL) {
        if (strcmp(temp->key, key) == 0) {
            return temp->value;
        }
        temp = temp->next;
    }
    return -1;
}
void delete(Node* table[], char* key) {
    unsigned int index = hashFunction(key);
    Node* temp = table[index];
    Node* prev = NULL;
    while (temp != NULL && strcmp(temp->key, key) != 0) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) {
        return;
    }
    if (prev == NULL) {
        table[index] = temp->next;
    } else {
        prev->next = temp->next;
    }
    free(temp->key);
    free(temp);
```

```
}
void printTable(Node* table[]) {
    for (int i = 0; i < TABLE_SIZE; i++) {
        Node* temp = table[i];
        printf("Index %d: ", i);
        while (temp != NULL) {
            printf("(%s, %d) -> ", temp->key, temp->value);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}
int main() {
    Node* hashTable[TABLE_SIZE] = {NULL};
    insert(hashTable, "apple", 1);
    insert(hashTable, "banana", 2);
    insert(hashTable, "orange", 3);
    insert(hashTable, "grape", 4);
    insert(hashTable, "cherry", 5);
    printTable(hashTable);
    printf("Search for 'apple': %d\n", search(hashTable, "apple"));
    printf("Search for 'banana': %d\n", search(hashTable, "banana"));
    delete(hashTable, "banana");
    printTable(hashTable);
    return 0;
}
```

Detecting Duplicates within a Dataset

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 100
typedef struct Node {
    int key;
    struct Node* next;
} Node;
Node* createNode(int key) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->next = NULL;
    return newNode;
}
unsigned int hashFunction(int key) {
    return key % TABLE_SIZE;
}
int insert(Node* table[], int key) {
    unsigned int index = hashFunction(key);
    Node* temp = table[index];
    while (temp != NULL) {
        if (temp->key == key) {
            return 1;
        }
        temp = temp->next;
    }
    Node* newNode = createNode(key);
    newNode->next = table[index];
    table[index] = newNode;
    return 0; // No duplicate
}
```

```
int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 3};
    int size = sizeof(arr) / sizeof(arr[0]);
    Node* hashTable[TABLE_SIZE] = {NULL};
    for (int i = 0; i < size; i++) {
        if (insert(hashTable, arr[i])) {
            printf("Duplicate found: %d\n", arr[i]);
            return 0;
        }
    }
    printf("No duplicates found\n");
    return 0;
}
```

Finding the Most Frequent Element

```
#include <stdio.h>
#include <stdlib.h>
#define TABLE_SIZE 100
typedef struct Node {
    int key;
    int count;
    struct Node* next;
} Node;
Node* createNode(int key) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->count = 1;
    newNode->next = NULL;
    return newNode;
}
```

```
unsigned int hashFunction(int key) {
    return key % TABLE_SIZE;
}
void insert(Node* table[], int key) {
    unsigned int index = hashFunction(key);
    Node* temp = table[index];
    while (temp != NULL) {
        if (temp->key == key) {
            temp->count++;
            return;
        }
        temp = temp->next;
    }
    Node* newNode = createNode(key);
    newNode->next = table[index];
    table[index] = newNode;
}
int findMostFrequent(Node* table[]) {
    int maxCount = 0;
    int mostFrequent = -1;
    for (int i = 0; i < TABLE_SIZE; i++) {
        Node* temp = table[i];
        while (temp != NULL) {
            if (temp->count > maxCount) {
                maxCount = temp->count;
                mostFrequent = temp->key;
            }
            temp = temp->next;
        }
    }
```

```
    return mostFrequent;
}
int main() {
    int arr[] = {1, 2, 3, 4, 5, 2, 3, 3};
    int size = sizeof(arr) / sizeof(arr[0]);
    Node* hashTable[TABLE_SIZE] = {NULL};

    for (int i = 0; i < size; i++) {
        insert(hashTable, arr[i]);
    }

    int mostFrequent = findMostFrequent(hashTable);
    printf("Most frequent element: %d\n", mostFrequent);

    return 0;
}
```