**Priority Queue Using Heap Max**

#include <stdio.h>

int tree\_array\_size = 20;

int heap\_size = 0;

const int INF = 100000;

void swap( int \*a, int \*b ) {

int t;

t = \*a;

\*a = \*b;

\*b = t;

}

//function to get right child of a node of a tree

int get\_right\_child(int A[], int index) {

if((((2\*index)+1) < tree\_array\_size) && (index >= 1))

return (2\*index)+1;

return -1;

}

//function to get left child of a node of a tree

int get\_left\_child(int A[], int index) {

if(((2\*index) < tree\_array\_size) && (index >= 1))

return 2\*index;

return -1;

}

//function to get the parent of a node of a tree

int get\_parent(int A[], int index) {

if ((index > 1) && (index < tree\_array\_size)) {

return index/2;

}

return -1;

}

void max\_heapify(int A[], int index) {

int left\_child\_index = get\_left\_child(A, index);

int right\_child\_index = get\_right\_child(A, index);

// finding largest among index, left child and right child

int largest = index;

if ((left\_child\_index <= heap\_size) && (left\_child\_index>0)) {

if (A[left\_child\_index] > A[largest]) {

largest = left\_child\_index;

}

}

if ((right\_child\_index <= heap\_size && (right\_child\_index>0))) {

if (A[right\_child\_index] > A[largest]) {

largest = right\_child\_index;

}

}

// largest is not the node, node is not a heap

if (largest != index) {

swap(&A[index], &A[largest]);

max\_heapify(A, largest);

}

}

void build\_max\_heap(int A[]) {

int i;

for(i=heap\_size/2; i>=1; i--) {

max\_heapify(A, i);

}

}

int maximum(int A[]) {

return A[1];

}

int extract\_max(int A[]) {

int maxm = A[1];

A[1] = A[heap\_size];

heap\_size--;

max\_heapify(A, 1);

return maxm;

}

void increase\_key(int A[], int index, int key) {

A[index] = key;

while((index>1) && (A[get\_parent(A, index)] < A[index])) {

swap(&A[index], &A[get\_parent(A, index)]);

index = get\_parent(A, index);

}

}

void decrease\_key(int A[], int index, int key) {

A[index] = key;

max\_heapify(A, index);

}

void insert(int A[], int key) {

heap\_size++;

A[heap\_size] = -1\*INF;

increase\_key(A, heap\_size, key);

}

void print\_heap(int A[]) {

int i;

for(i=1; i<=heap\_size; i++) {

printf("%d\n",A[i]);

}

printf("\n");

}

int main() {

int A[tree\_array\_size];

insert(A, 20);

insert(A, 15);

insert(A, 8);

insert(A, 10);

insert(A, 5);

insert(A, 7);

insert(A, 6);

insert(A, 2);

insert(A, 9);

insert(A, 1);

print\_heap(A);

increase\_key(A, 5, 22);

print\_heap(A);

decrease\_key(A, 1, 13);

print\_heap(A);

printf("%d\n\n", maximum(A));

printf("%d\n\n", extract\_max(A));

print\_heap(A);

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

printf("%d\n", extract\_max(A));

return 0;

}

**Priority Queue Using Heap Min**

#include <stdio.h>

int tree\_array\_size = 20;

int heap\_size = 0;

const int INF = 100000;

void swap( int \*a, int \*b ) {

int t;

t = \*a;

\*a = \*b;

\*b = t;

}

//function to get right child of a node of a tree

int get\_right\_child(int A[], int index) {

if((((2\*index)+1) < tree\_array\_size) && (index >= 1))

return (2\*index)+1;

return -1;

}

//function to get left child of a node of a tree

int get\_left\_child(int A[], int index) {

if(((2\*index) < tree\_array\_size) && (index >= 1))

return 2\*index;

return -1;

}

//function to get the parent of a node of a tree

int get\_parent(int A[], int index) {

if ((index > 1) && (index < tree\_array\_size)) {

return index/2;

}

return -1;

}

void min\_heapify(int A[], int index) {

int left\_child\_index = get\_left\_child(A, index);

int right\_child\_index = get\_right\_child(A, index);

// finding smallest among index, left child and right child

int smallest = index;

if ((left\_child\_index <= heap\_size) && (left\_child\_index>0)) {

if (A[left\_child\_index] < A[smallest]) {

smallest = left\_child\_index;

}

}

if ((right\_child\_index <= heap\_size && (right\_child\_index>0))) {

if (A[right\_child\_index] < A[smallest]) {

smallest = right\_child\_index;

}

}

// smallest is not the node, node is not a heap

if (smallest != index) {

swap(&A[index], &A[smallest]);

min\_heapify(A, smallest);

}

}

void build\_min\_heap(int A[]) {

int i;

for(i=heap\_size/2; i>=1; i--) {

min\_heapify(A, i);

}

}

int minimum(int A[]) {

return A[1];

}

int extract\_min(int A[]) {

int minm = A[1];

A[1] = A[heap\_size];

heap\_size--;

min\_heapify(A, 1);

return minm;

}

void decrease\_key(int A[], int index, int key) {

A[index] = key;

while((index>1) && (A[get\_parent(A, index)] > A[index])) {

swap(&A[index], &A[get\_parent(A, index)]);

index = get\_parent(A, index);

}

}

void increase\_key(int A[], int index, int key) {

A[index] = key;

min\_heapify(A, index);

}

void insert(int A[], int key) {

heap\_size++;

A[heap\_size] = INF;

decrease\_key(A, heap\_size, key);

}

void print\_heap(int A[]) {

int i;

for(i=1; i<=heap\_size; i++) {

printf("%d\n",A[i]);

}

printf("\n");

}

int main() {

int A[tree\_array\_size];

insert(A, 20);

insert(A, 15);

insert(A, 8);

insert(A, 10);

insert(A, 5);

insert(A, 7);

insert(A, 6);

insert(A, 2);

insert(A, 9);

insert(A, 1);

print\_heap(A);

increase\_key(A, 5, 22);

print\_heap(A);

printf("%d\n\n", minimum(A));

printf("%d\n\n", extract\_min(A));

print\_heap(A);

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

printf("%d\n", extract\_min(A));

return 0;

}