

# Data Structures and Algorithms

## Exercise 5

Markus Innerebner, Romans Kasperovics  
dsa(a)inf.unibz.it

Free University of Bozen - Bolzano, Italy

Thu, April 02, 2009

Write a C function `LE* ExtractLess(LE** List, int Max)` that extracts from a double-linked list all elements with key values less than `Max`, and returns a new double-linked list with extracted elements. No additional memory should be allocated.

```
#include <stdio.h>
#include <stdlib.h>
#define N 10

typedef struct le {
    int key;
    struct le* prev;
    struct le* next;
} LE;
```

# Solution 05

## Task 1: Auxiliary Functions

```
void Print(LE* list) {  
    for ( ; list  $\neq$  NULL; list = list→next)  
        printf("%d ", list→key);  
    printf("\n");  
}  
LE* Insert(LE* list, int key) {  
    LE* tmp = (LE*) malloc(sizeof(LE));  
    tmp→key = key; tmp→prev = NULL; tmp→next = list;  
    if (list  $\neq$  NULL ) list→prev = tmp;  
    return tmp;  
}  
void Delete(LE** list) {  
    while (*list  $\neq$  NULL) {  
        LE* tmp = *list;  
        *list = (*list)→next;  
        free(tmp);  
    }  
}
```

# Solution 05

## Task 1: ExtractLess

```
LE* ExtractLess(LE** List, int Max) {
    LE *lit = *List, *res = NULL, *rit = NULL;
    while (lit != NULL) {
        if (lit->key < Max) {
            if (lit->prev != NULL) lit->prev->next = lit->next;
            else *List = lit->next;
            if (lit->next != NULL) lit->next->prev = lit->prev;
            if (rit == NULL) {
                res = lit;
                res->prev = NULL;
            } else rit->next = lit;
            rit = lit;
            lit = lit->next;
            rit->next = NULL;
        } else lit = lit->next;
    }
    return res;
}
```

# Solution 05

## Task 1: Main Function

```
int main() {
    int A[N] = {2, 14, 56, 98, 4, 34, 18, 1, 0, 88};
    LE* head = NULL; // points to "nowhere"
    int i;
    for (i=N-1; i>=0; i--)
        head = Insert(head, A[i]);
    Print(head);
    LE* newlist = ExtractLess(&head, 30);
    Print(head);
    Print(newlist);
    Delete(&head);
    Delete(&newlist);
    return 0;
}
```

Consider a heap-based implementation of a priority queue. Write a function `void UpdateQueue(int A[], int n, int i, int p)` that for the given array  $A$  of size  $n$ , index  $i$ , and a new priority  $p$ , changes the priority of  $A[i]$  to  $p$ . If after the assignment, the array  $A$  does not hold the heap properties, the function reorders  $A$  so that the heap properties are restored. Your function should run in  $O(\lg n)$  time.

```
#include <stdio.h>
#define parent(i) ((i-1)/2)
#define lchild(i) ((i+1)*2-1)
#define rchild(i) ((i+1)*2)
#define swap(x, y) {int tmp = x; x = y; y = tmp;}
#define N 10
```

# Solution 05

## Task 2: PrintQueue + Heapify

```
void PrintQueue(int A[], int n) {  
    int i;  
    for (i=0; i<n; i++)  
        printf("%d ",A[i]);  
    printf("\n");  
}  
void heapify (int A[], int n, int i) {  
    int l = lchild(i), r = rchild(i);  
    int max = i;  
    if ( l<n && A[l]>A[max] ) max = l;  
    if ( r<n && A[r]>A[max] ) max = r;  
    if ( max≠i ) {  
        swap(A[i],A[max]);  
        heapify(A,n,max);  
    }  
}
```

Observation: the new element can either go only up or only down, and no up / down combinations. Thus, the running time is  $O(\lg n)$

```
void UpdateQueue (int A[], int n, int i, int p) {  
    // going up  
    while (i > 0 && A[parent(i)] < p) {  
        A[i] = A[parent(i)];  
        i = parent(i);  
    }  
    A[i] = p;  
    // going down  
    heapify(A,n,i);  
}
```

# Solution 05

## Task 2: Main Function

```
int main() {  
    int Q[N]={12,11,9,7,10,8,5,6,3,4};  
    PrintQueue(Q,N);  
    UpdateQueue(Q, N, 3, 17);  
    PrintQueue(Q,N);  
    return 0;  
}
```